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 Lecture): 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

2nd Lec

   - 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

Examples

1st Lec

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

2nd Lec

Compiler Front End

After scanning and parsing:

- Semantic analysis enforces static semantics:
  - Identification (symbol table)
  - Type checking
- Context-sensitive static semantics cannot be specified by a context-free grammar (CFG) (Lecture 3)
- An attribute grammar augments a CFG to complete the specification of what legal programs should look like

Context-Sensitive Restrictions (Static Semantics)

- Is x a variable, method, array, class or package?
- Is x declared before used?
- Which declaration of x does this reference (identification)?
- Is an expression type-consistent?
- Does the dimension of an array match with the declaration?
- Is an array reference in bounds?
- Is a method called with the right number and types of args?
- Is break or continue enclosed in a loop construct?
- etc.

These cannot be specified using a CFG!

Examples: VC Programs

Program 1: ``a'' used but not declared

```
void main() {
  a = 3;
}
```

Program 2: ``f'' called with the wrong number of arguments

```
int f(int i) { }
void main() {
  f(1, 2);
}
```

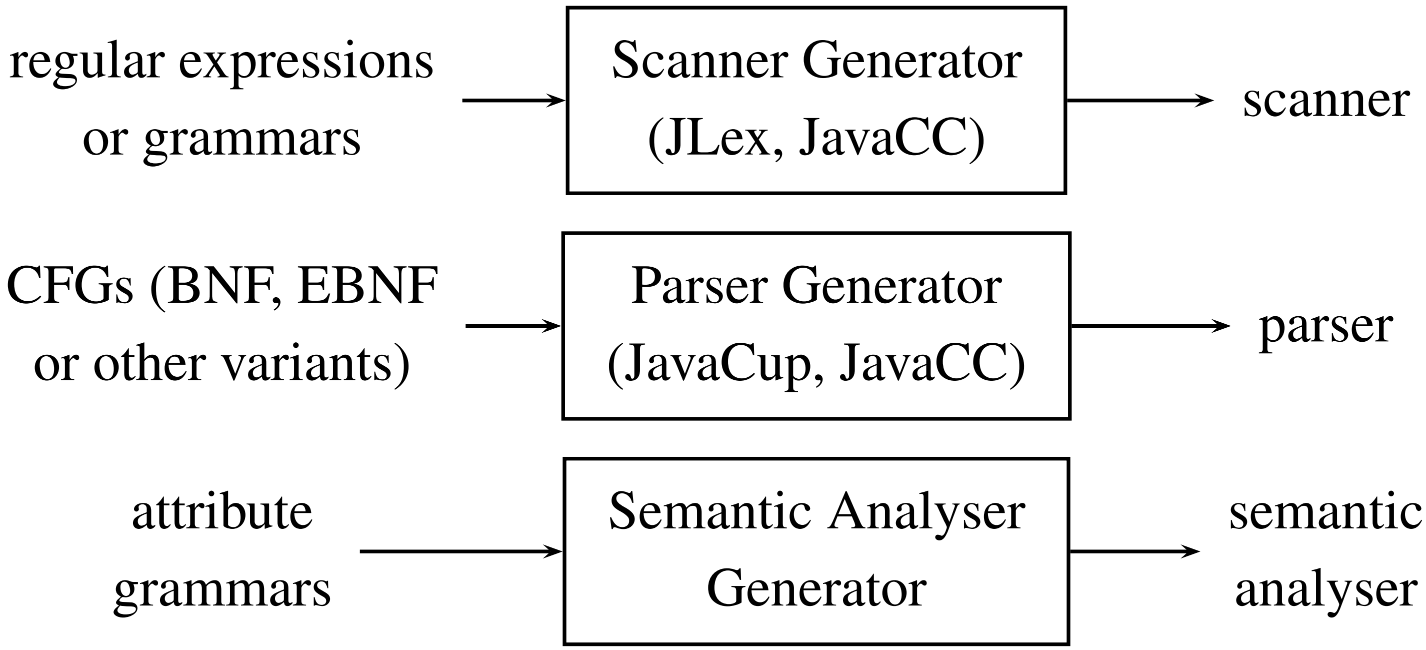
Program 3: incompatible operands

```
void main() {
  int i;
  i = true + 1;
}
```

Context-Sensitive Analysis (Semantic Analysis)

- Ad hoc techniques
  - Symbol table and codes
  - “Action routines” in parser generators
- Formal methods:
  - Attribute grammars (or other variants)
  - Type systems and checking algorithms
- Our approach for VC:
  - Static semantics specified
    - in English (the VC Language Specification), and
    - by an attribute grammar in part
  - Build a semantic analyser by hand

Automatic Construction of the Front End



There are no widely acceptable semantic analysers.

Attribute Grammars: Informal Definition

- Attribute grammars:
  - Generalisation of CFGs
  - Each attribute associated with a grammar symbol
  - Each semantic rule associated with a production defining attributes
  - High-level spec, independent of any evaluation order
- Dependences between attributes
  - Attributes computed from other attributes
  - Synthesised attributes: computed from children
  - Inherited attributes: computed from parent and siblings

Attribute Grammars: Formal Definition

An attribute grammar is a triple:

$$A = (G, V, F)$$

where

- G is a CFG,
- V is a finite set of distinct attributes, and
- F is a finite set of semantic rules about the attributes.

Note:

- Each attribute is associated with a grammar symbol
- Each semantic rule is associated with a production that makes reference only to the attributes associated with the symbols in the production (the effect of a rule is localised within the production)

Attributes Associated with a Grammar Symbol

- A attribute can represent anything we choose:
  - a string
  - a number
  - a type
  - a memory location
  - a piece of code
  - etc.
- Each attribute has a name and a type

Example 1: (Simplified) Expression Grammar

- The grammar:

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

- The grammar is ambiguous but can be used to specify the static semantics if the parse or syntax tree has been built using an unambiguous grammar (Lecture 6)!

- Assume that
  - ”/” is left-associative, and
  - Mixed expressions are promoted to float-point ones throughout (Example: 5/2/2.0 evaluated to 1.25 not 1.00)

- Give an attribute grammar for the value of an expression

Attribute Grammar for Example 1

Production			Semantic Rules
$\langle$S$\rangle$	$\rightarrow$	$\langle$expr$\rangle$	$\langle$expr.type$\rangle$ = if $\langle$expr.isFloat$\rangle$ then float else int $\langle$S.val$\rangle$ = $\langle$expr.val$\rangle$
$\langle$expr$_1\rangle$	$\rightarrow$	$\langle$expr$_2\rangle$ / $\langle$expr$_3\rangle$	$\langle$expr$_1$.isFloat$\rangle$ = $\langle$expr$_2$.isFloat$\rangle$ or $\langle$expr$_3$.isFloat$\rangle$ $\langle$expr$_2$.type$\rangle$ = $\langle$expr$_1$.type$\rangle$ $\langle$expr$_3$.type$\rangle$ = $\langle$expr$_1$.type$\rangle$ $\langle$expr$_1$.val$\rangle$ = if ($\langle$expr$_1$.type$\rangle$ == int) then $\langle$expr$_2$.val$\rangle$ **div** $\langle$expr$_3$.val$\rangle$ else $\langle$expr$_2$.val$\rangle$ / $\langle$expr$_3$.val$\rangle$
$\langle$expr$\rangle$	$\rightarrow$	**num**	$\langle$expr.isFloat$\rangle$ = false $\langle$expr.val$\rangle$ = if ($\langle$expr.type$\rangle$ == int) then **num**.val else Float(**num**.val)
$\langle$expr$\rangle$	$\rightarrow$	**num . num**	$\langle$expr.isFloat$\rangle$ = true $\langle$expr.val$\rangle$ = **num.num**.val

Attribute Grammar for Example 1

Production			Semantic Rules
$\langle S \rangle$	$\rightarrow$	$\langle expr \rangle$	
$\langle expr_1 \rangle$	$\rightarrow$	$\langle expr_2 \rangle$ / $\langle expr_3 \rangle$	$\langle expr_1.isFloat \rangle$ = $\langle expr_2.isFloat \rangle$ or $\langle expr_3.isFloat \rangle$
$\langle expr \rangle$	$\rightarrow$	**num**	$\langle expr.isFloat \rangle$ = false
$\langle expr \rangle$	$\rightarrow$	**num . num**	$\langle expr.isFloat \rangle$ = true

Attribute Grammar for Example 1

Production	Semantic Rules
$\langle S \rangle \rightarrow \langle \text{expr} \rangle$	$\langle \text{expr.type} \rangle$ = if $\langle \text{expr.isFloat} \rangle$ then float else int $\langle \text{S.val} \rangle = \langle \text{expr.val} \rangle$
$\langle \text{expr}_1 \rangle \rightarrow \langle \text{expr}_2 \rangle$ / $\langle \text{expr}_3 \rangle$	$\langle \text{expr}_2\text{.type} \rangle = \langle \text{expr}_1\text{.type} \rangle$ $\langle \text{expr}_3\text{.type} \rangle = \langle \text{expr}_1\text{.type} \rangle$
$\langle \text{expr} \rangle \rightarrow$ **num**	
$\langle \text{expr} \rangle \rightarrow$ **num . num**	

Attribute Grammar for Example 1

Production			Semantic Rules
$\langle S \rangle$	$\rightarrow$	$\langle expr \rangle$	$\langle S.val \rangle = \langle expr.val \rangle$
$\langle expr_1 \rangle$	$\rightarrow$	$\langle expr_2 \rangle$ / $\langle expr_3 \rangle$	$\langle expr_1.val \rangle$ = if ($\langle expr_1.type \rangle$ == int) then $\langle expr_2.val \rangle$ **div** $\langle expr_3.val \rangle$ else $\langle expr_2.val \rangle$ / $\langle expr_3.val \rangle$
$\langle expr \rangle$	$\rightarrow$	**num**	$\langle expr.val \rangle$ = if ($\langle expr.type \rangle$ == int) then **num**.val else Float(**num**.val)
$\langle expr \rangle$	$\rightarrow$	**num . num**	$\langle expr.val \rangle$ = **num.num**.val

Attribute Grammar for Example 1 (Cont'd)

- div: integer division
- /: floating-point division
- Float: a function converting an integer to a float value
- num.val and num.num.val
  - computed by the scanner before semantic analysis begins
  - known as (intrinsic) synthesised attributes

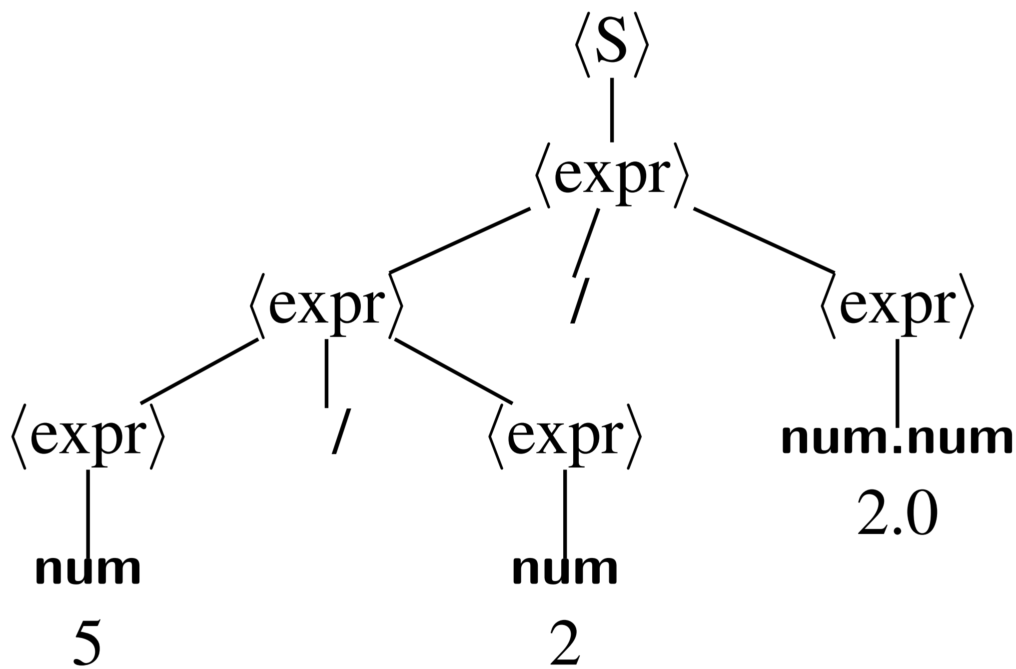
Attribute Grammar for Example 1 (Cont'd)

- Synthesised attribute isFloat over {true, false} – indicating if any part of a subexpression has a floating-point value
- Inherited attribute type over {int, float} – giving the type of each subexpression
- Synthesised attribute val of the type int – giving the value of each subexpression
- Dependences between the attributes:

  $$\texttt{isFloat} \longrightarrow \texttt{type} \longrightarrow \texttt{val}$$

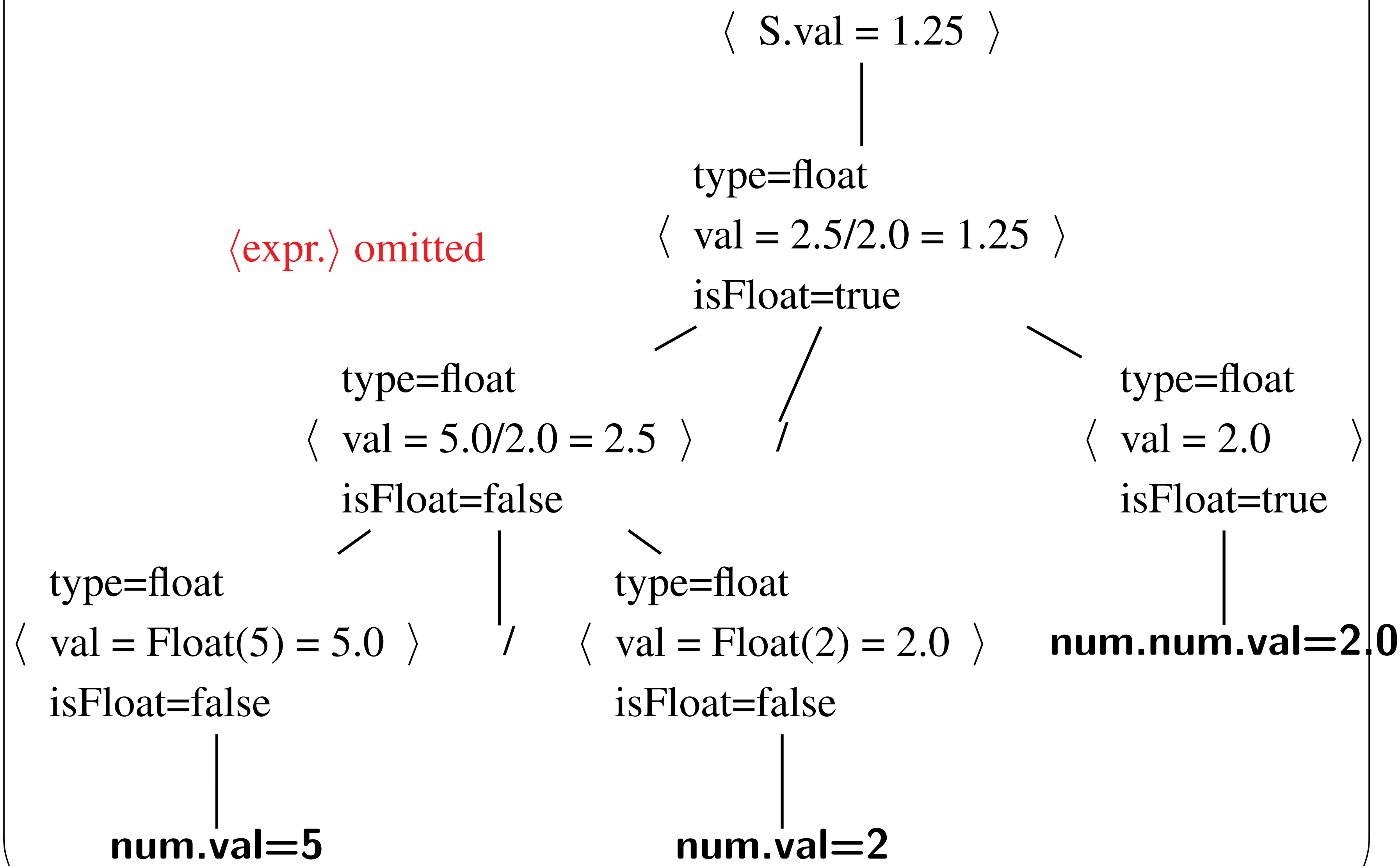
  I.e., val depends on type, which depends on isFloat

Parse Tree for **5/2/2.0**

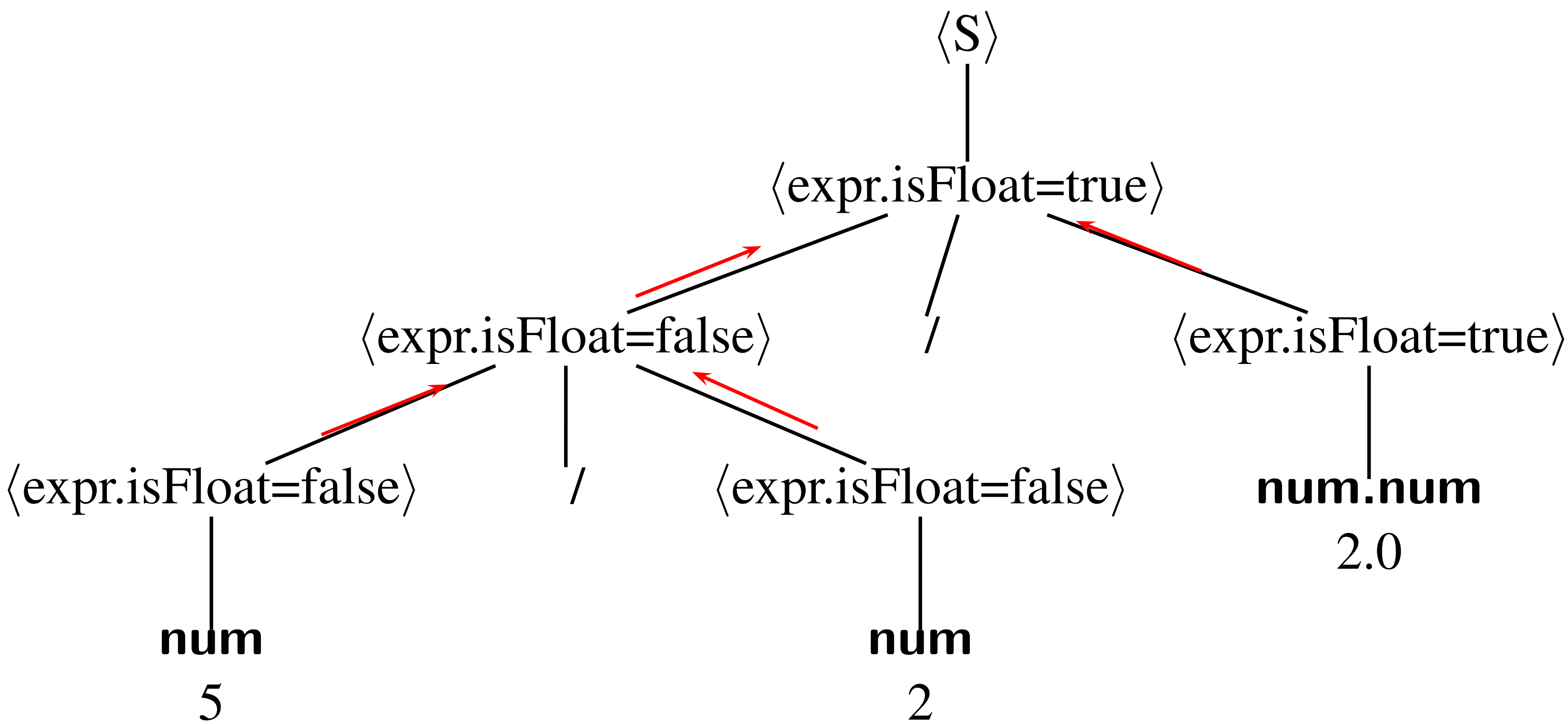


- "/" is assumed to be left-associative
- The other tree (making "/" right-associative) not used

Decorated (Annotated Parse Tree)

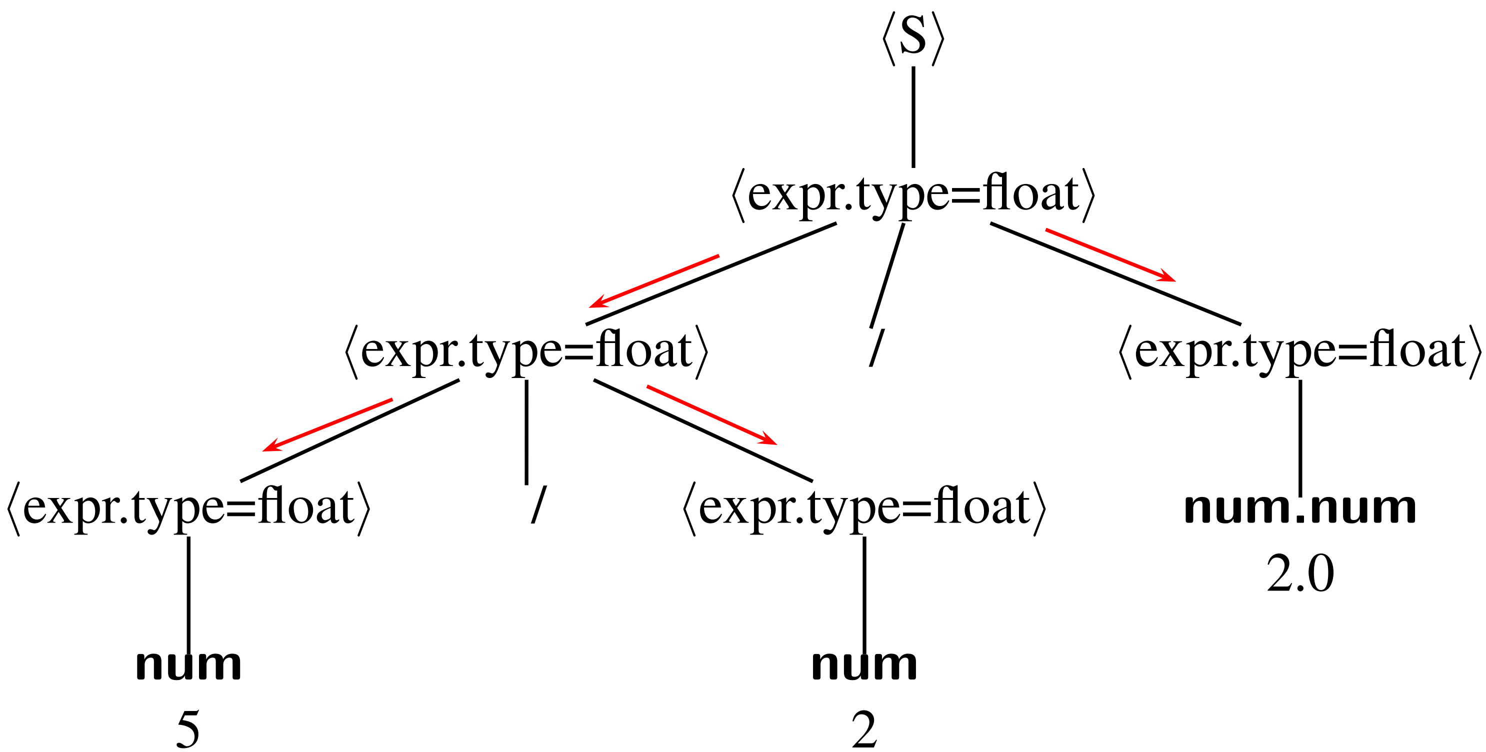


The flow of Synthesised Attribute isFloat



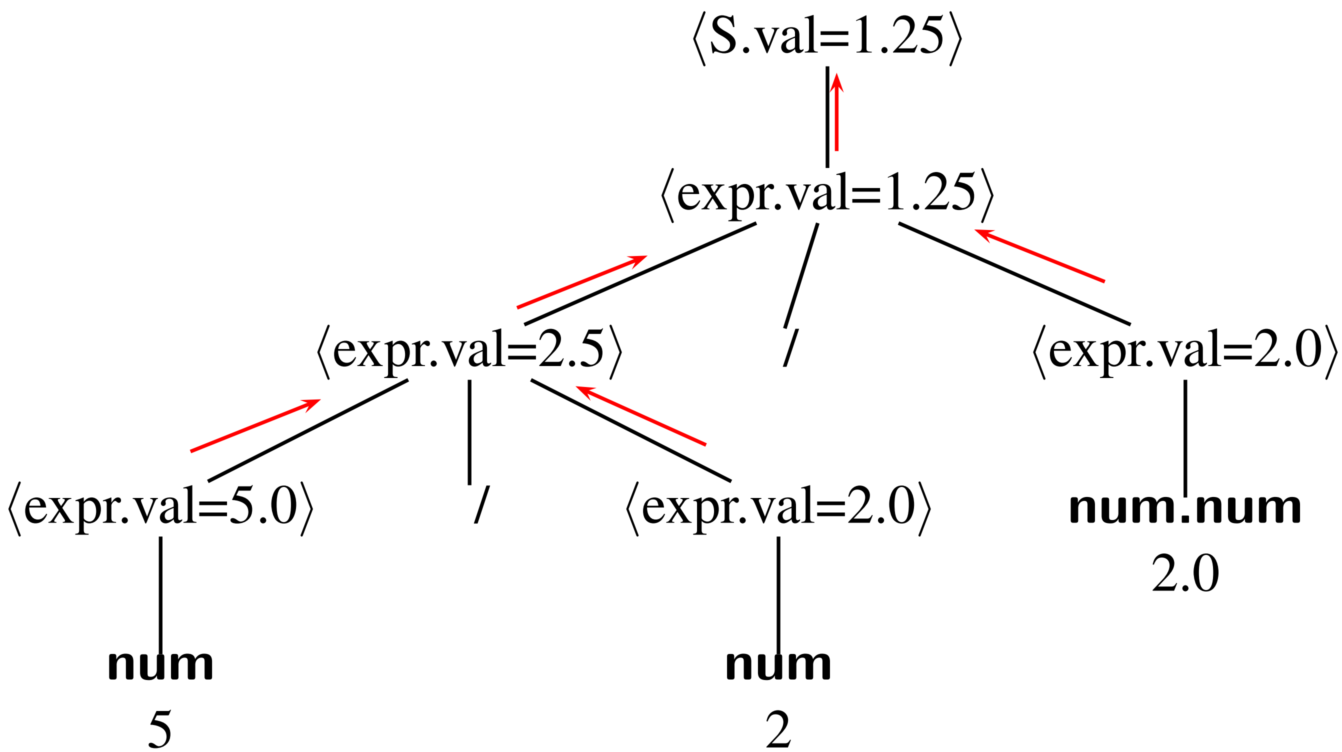
The synthesised attributes flow bottom-up

The flow of Inherited Attribute type



The inherited attributes flow top-down (or left-to-right or right-to-left)

The flow of Synthesised Attribute val



The synthesised attributes flow bottom-up

Formal Definitions of Synthesised and Inherited Attributes

Let

- $X_0 \rightarrow X_1 X_2 \cdots X_n$ be a production, and
- $A(X)$ be the set of attributes associated with a grammar symbol X

Then

- A synthesised attribute, syn, of X_0 is computed by:

$$X_0.syn \quad = \quad f(A(X_1), A(X_2), \ldots, A(X_n))$$

  syn on a tree node depends on those on its children

- An inherited attribute, inh, of X_i, where $1 \leqslant i \leqslant n$, is computed by:

$$X_i.inh \quad = \quad g(A(X_0), A(X_1), \ldots, A(X_n))$$

  – inh on a tree node depends on those on its parent and/or siblings
  – $X_i.inh$ can depend on the other attributes in $A(X_i)$

Attribute Evaluators

- Tree Walkers: Traverse the parse or syntax tree in one pass or multiple passes at compile time
  - Capable of evaluating any noncircular attribute grammar
  - An attribute grammar is circular if an attribute depends on itself
  - Can decide the circularity in exponential time
  - Too complex to be used in practice
- Rule-Based Methods: The compiler writer analyses the attribute grammar and fixes an evaluation order at compiler-construction time
  - Trees can still be used for attribute evaluation
  - Almost all reasonable grammars can be handled this way
  - Used practically in all compilers

A Non-Circular Attribute Grammar Evaluator

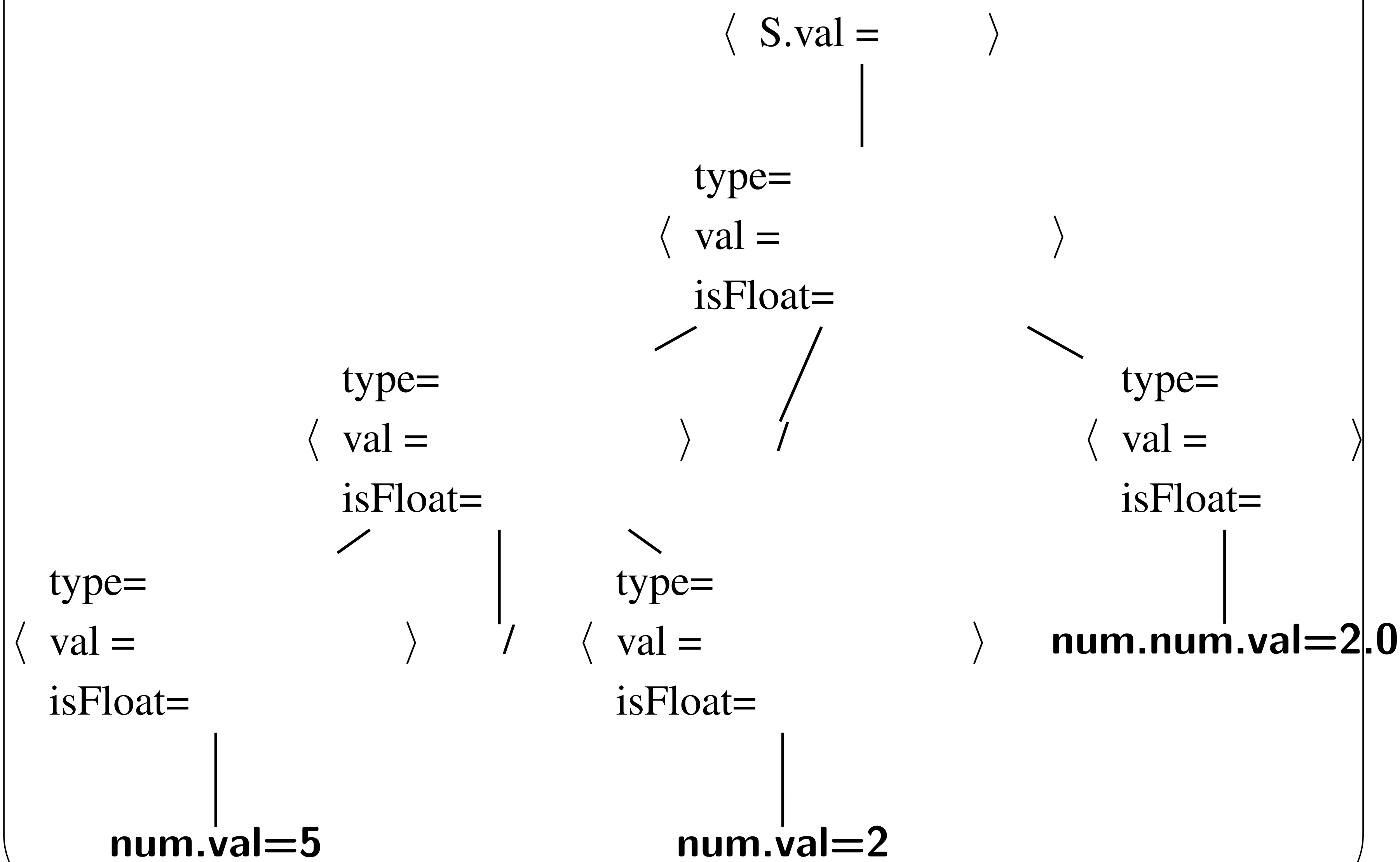
```
while ( attributes remain to be evaluated ) {
  visitNode(S) // S is the start symbol
}
void visitNode (AST N) { // i.e., ProcessNode(AST N)
  if ( N is a nonterminal ) { // N -> X1 X2 ... Xm
    for (i = 1; i <= m; i++)  {
      if ( Xi is nonterminal ) {
        Evaluate all possible inherited attributes of Xi
        visitNode(Xi)
      }
    }
  }
  Evaluate all possible synthesised attributes of N
}
```

- Pre-order visits: propagate inherited attributes downwards
- Post-order visits: propagate synthesised attributes upwards
- At least one attribute will be evaluated in one pass. The worst-case time complexity is $O(n^2)$, where n is #nodes (assuming that the number of attributes is $O(n)$.)

A Trace of visitNode on Example 1

- Pass 1: the computation of $\langle$isFloat$\rangle$
- Pass 2: the computation of $\langle$type$\rangle$ and $\langle$val$\rangle$
- Two passes required for evaluating all three attributes

A Trace of visitNode on Example 1



Reading

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

Next Class: Attribute Grammars (Cont'd)