

Semantic Analysis

CSE 307 – Principles of Programming Languages

Slides courtesy:

Prof. Paul Fodor, Stony Brook University

Role of Semantic Analysis

- Syntax vs. Semantics:
 - syntax concerns the form of a valid program (described conveniently by a context-free grammar CFG)
 - semantics concerns its meaning: rules that go beyond mere form (e.g., the number of arguments contained in a call to a subroutine matches the number of formal parameters in the subroutine definition):
 - Defines what the program means
 - Detects if the program is correct
 - Helps to translate it into another representation

Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are:
 - semantic analysis
 - (intermediate) code generation
- Semantic rules are divided into:
 - *static* semantics enforced at compile time
 - *dynamic* semantics: the compiler generates code to enforce dynamic semantic rules at run time (or calls libraries to do it) (for errors like division by zero, out-of-bounds index in array)
- The principal job of the *semantic analyzer* is to enforce static semantic rules, plus:
 - constructs a syntax tree
 - information gathered is needed by the code generator

Role of Semantic Analysis

- Parsing, semantic analysis, and intermediate code generation are interleaved:
 - a common approach interleaves parsing construction of a syntax tree with phases for semantic analysis and code generation
 - The semantic analysis and intermediate code generation **annotate** the parse tree with *attributes*
 - *Attribute grammars* provide a formal framework for the decoration of a syntax tree
 - The *attribute flow* constrains the order(s) in which nodes of a tree can be decorated.
 - replaces the parse tree with a syntax tree that reflects the input program in a more straightforward way

Role of Semantic Analysis

- Dynamic checks: semantic rules enforced at run time
 - C requires no dynamic checks at all (it relies on the hardware to find division by zero, or attempted access to memory outside the bounds of the program).
 - Java check as many rules as possible, so that an untrusted program cannot do anything to damage the memory or files of the machine on which it runs.
- Many compilers that generate code for dynamic checks provide the option of disabling them (enabled during program development and testing, but disables for production use, to increase execution speed)
 - Hoare: “*like wearing a life jacket on land, and taking it off at sea*”

Role of Semantic Analysis

- **Assertions**: logical formulas written by the programmers regarding the values of program data used to reason about the correctness of their algorithms (the assertion is expected to be **true** when execution reaches a certain point in the code):
 - Java: `assert denominator != 0;`
 - An **AssertionError** exception will be thrown if the semantic check fails at run time.
 - C: `assert (denominator != 0) ;`
 - If the assertion fails, the program will terminate abruptly with a message: `a.c:10: failed assertion 'denominator != 0'`
 - Some languages also provide explicit support for *invariants*, **preconditions**, and **post-conditions**.
 - Like Dafny from Microsoft <https://github.com/Microsoft/dafny>

Java Assertions

- Java example:
 - An assertion in Java is a statement that enables us to assert an assumption about our program.
 - An assertion contains a Boolean expression that should be true during program execution.
 - Assertions can be used to assure program correctness and avoid logic errors.
 - An assertion is declared using the Java keyword **assert** in JDK 1.5 as follows:

assert assertion; //OR

assert assertion : detailMessage;

where **assertion** is a Boolean expression and **detailMessage** is a primitive-type or an **Object** value.

Java Assertion Example

```
public class AssertionDemo {  
    public static void main(String[] args) {  
        int i;  
        int sum = 0;  
        for (i = 0; i < 10; i++) {  
            sum += i;  
        }  
        assert i==10;  
        assert sum>10 && sum<5*10 : "sum is " + sum;  
    }  
}
```

- When an assertion statement is executed, Java evaluates the assertion
 - If it is false, an **AssertionError** will be thrown

Java Assertion Example

- The **AssertionError** class has a no-arg constructor and seven overloaded single-argument constructors of type **int**, **long**, **float**, **double**, **boolean**, **char**, and **Object**
 - For the first assert statement in the example (with no detail message), the no-arg constructor of **AssertionError** is used.
 - For the second assert statement with a detail message, an appropriate **AssertionError** constructor is used to match the data type of the message.
 - Since **AssertionError** is a subclass of **Error**, when an assertion becomes false, the program displays a message on the console and exits

Running Programs with Assertions

- By default, the assertions are disabled at runtime
 - To enable it, use the switch **-enableassertions**, or **-ea** for short, as follows:

```
java -ea AssertionDemo  
public class AssertionDemo {  
    public static void main(String[] args) {  
        int i; int sum = 0;  
        for (i = 0; i < 10; i++) {  
            sum += i;  
        }  
        assert i!=10;  
    }  
}
```

Exception in thread "main" java.lang.AssertionError
at AssertionDemo.main([AssertionDemo.java:7](#))

Running Programs with Assertions

- Assertions can be selectively enabled or disabled at class level or package level
 - The disable switch is **-disableassertions** or **-da** for short.
 - For example, the following command enables assertions in package **package1** and disables assertions in class **Class1**:

```
java -ea:package1 -da:Class1 AssertionDemo
```

Using Exception Handling or Assertions?

- Assertion should not be used to replace exception handling.
 - Exception handling deals with unusual circumstances during program execution.
 - Assertions are to assure the correctness of the program
 - Exception handling addresses *robustness* and assertion addresses *correctness*
 - Assertions are used for internal consistency and validity checks
 - Assertions are checked at runtime and can be turned on or off at startup time

Using Exception Handling or Assertions?

- Do not use assertions for argument checking in public methods:
 - Valid arguments that may be passed to a public method are considered to be part of the method's contract
 - The contract must always be obeyed whether assertions are enabled or disabled
 - For example, the following code in the **Circle** class should be rewritten using exception handling:

```
public void setRadius(double newRadius) {  
    assert newRadius >= 0;  
    radius = newRadius;  
}
```

Using Exception Handling or Assertions?

- Use assertions to reaffirm assumptions.
 - This gives you more confidence to assure correctness of the program.
 - A common use of assertions is to replace assumptions with assertions in the code.
 - A good use of assertions is place assertions in a switch statement without a default case. For example:

```
switch (month) {  
    case 1: ... ; break;  
    case 2: ... ; break;  
    ...  
    case 12: ... ; break;  
    default: assert false : "Invalid month: " + month;  
}
```

Correctness of Algorithms

- **Loop *Invariants*:** used to prove correctness of a loop with respect to pre- and post-conditions

[Pre-condition for the loop]

while (G)

[Statements in the body of the loop]

end while

[Post-condition for the loop]

A loop is correct with respect to its pre- and post-conditions if, and only if, whenever the algorithm variables satisfy the pre-condition for the loop and the loop terminates after a finite number of steps, the algorithm variables satisfy the post-condition for the loop

Loop Invariant

- A **loop invariant $I(n)$** is a predicate with domain a set of integers, which for each iteration of the loop (**mathematical induction**), if the predicate is true before the iteration, then it is true after the iteration

If **the loop invariant $I(0)$ is true before the first iteration of the loop** AND

After a finite number of iterations of the loop, the guard G becomes false AND

The truth of **the loop invariant ensures the truth of the post-condition of the loop**

then the loop will be correct with respect to its pre- and post-conditions

Loop Invariant

- **Correctness of a Loop to Compute a Product:**

A loop to compute the product mx for a nonnegative integer m and a real number x , without using multiplication

[Pre-condition: m is a nonnegative integer, x is a real number, $i = 0$, and $\text{product} = 0$]

while ($i \neq m$)

$\text{product} := \text{product} + x$

$i := i + 1$

end while

[Post-condition: $\text{product} = mx$]

Loop invariant $I(n)$: $i = n$ and $\text{product} = n * x$

Guard G : $i \neq m$

Static analysis

- Static analysis: compile-time algorithms that predict run-time behavior
 - **Type checking**, for example, is static and precise in ML: the compiler ensures that no variable will ever be used at run time in a way that is inappropriate for its type
 - By contrast, languages like Lisp and Smalltalk accept the run-time overhead of dynamic type checks
 - In Java, type checking is mostly static, but dynamically loaded classes and type casts require run-time checks
- Static analysis is usually done for **Optimizations**
 - Optimizations can lead to security risks if implemented incorrectly (see 2018 Spectre hardware vulnerability: microarchitecture-level optimizations to code execution [can] leak information)

Attribute Grammars

- Both semantic analysis and (intermediate) code generation can be described in terms of *annotation*, or "*decoration*" of a parse or syntax tree
 - attributes are properties/actions attached to the production rules of a grammar
 - ATTRIBUTE GRAMMARS provide a formal framework for decorating a parse tree

More than just CFG

- The language $L = a^n b^n c^n$ (e.g., *abc*, *aabbcc*, *aaabbbccc*, ...) is not context free
- It can be captured, however, using an attribute grammar:

$G \rightarrow \mathbf{As\ Bs\ Cs} \triangleright G.ok := (As.val == Bs.val \wedge Bs.val == Cs.val)$

$As_1 \rightarrow a\ As_2 \triangleright As_1.val := As_2.val + 1$

$As \rightarrow \epsilon \triangleright As.val := 0$

$Bs_1 \rightarrow b\ Bs_2 \triangleright Bs_1.val := Bs_2.val + 1$

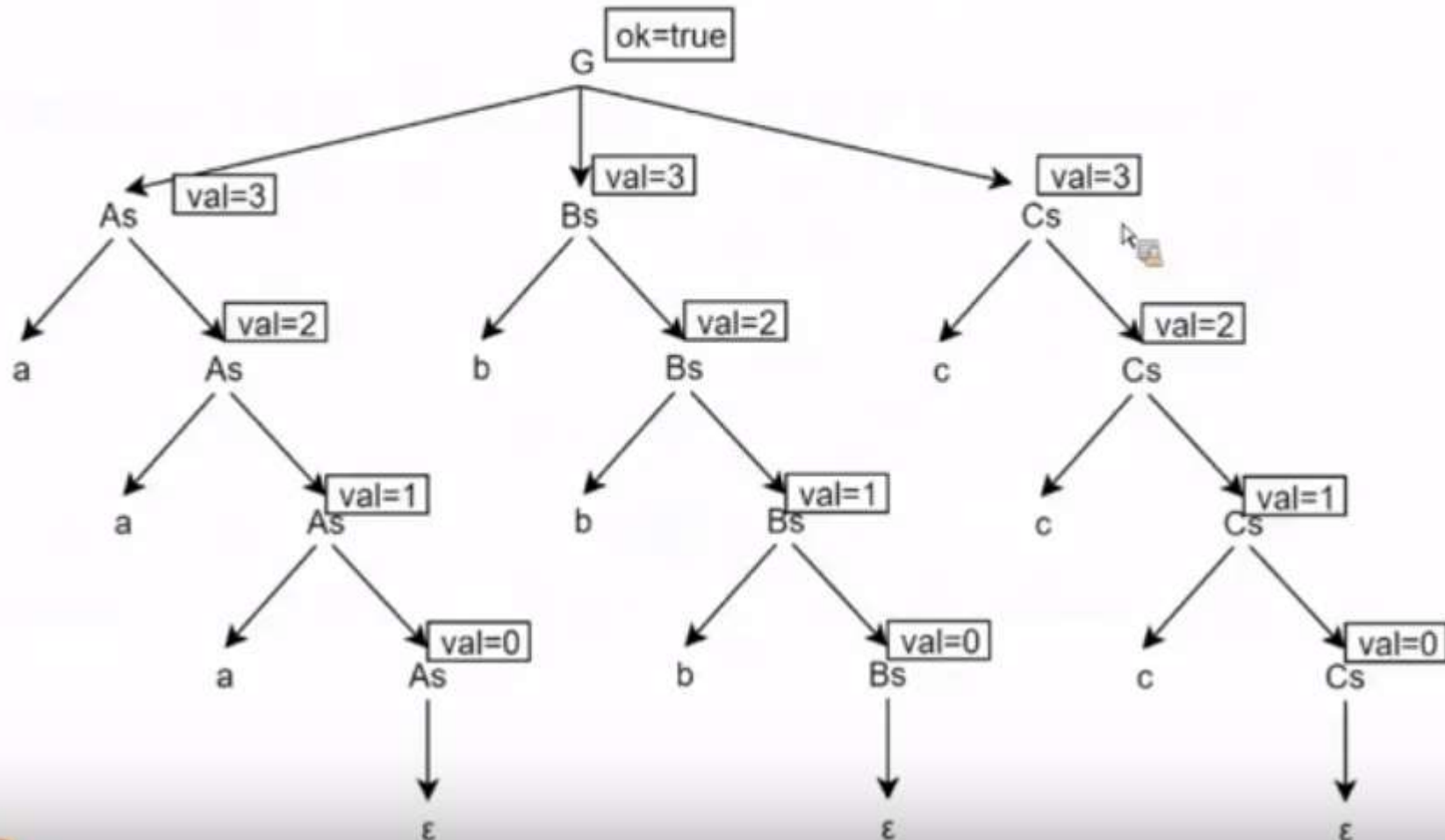
$Bs \rightarrow \epsilon \triangleright Bs.val := 0$

$Cs_1 \rightarrow c\ Cs_2 \triangleright Cs_1.val := Cs_2.val + 1$

$Cs \rightarrow \epsilon \triangleright Cs.val := 0$

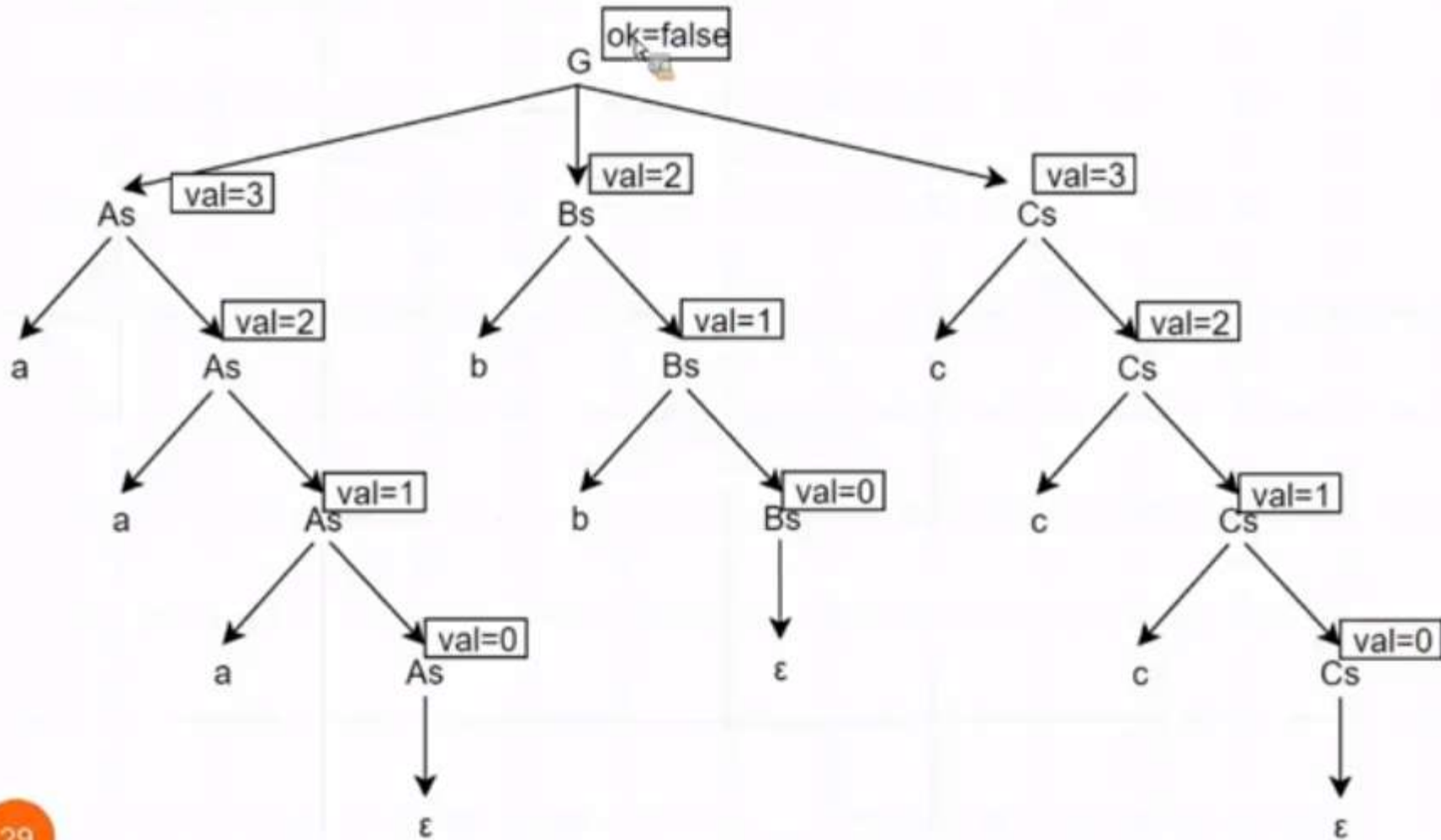
More than just CFG

- Annotate tree for "aaabbbbaaa":



More than just CFG

- Annotate tree for "aaabbaaa":



Attribute Grammars Example with variables

Tokens: `int (attr val), var (attr name)`

`S -> var = E`

▷ `assign(var.name, E.val)`

`E1 -> E2 + T`

▷ `E1.val = add(E2.val, T.val)`

`E1 -> E2 - T`

▷ `E1.val = sub(E2.val, T.val)`

`E -> T`

▷ `E.val = T.val`

`T -> var`

▷ `T.val = lookup(var.name)`

`T -> int`

▷ `T.val = int.val`

Input:

“bar = 50

foo = 100 + 200 – bar”

Evaluating Attributes

- *Synthesized Attributes:*
 - Data flows bottom-up
 - Can be parsed by LR grammars
- *Inherited Attributes:*
 - Data flows top-down and bottom-up
 - Can be parsed with LL grammars

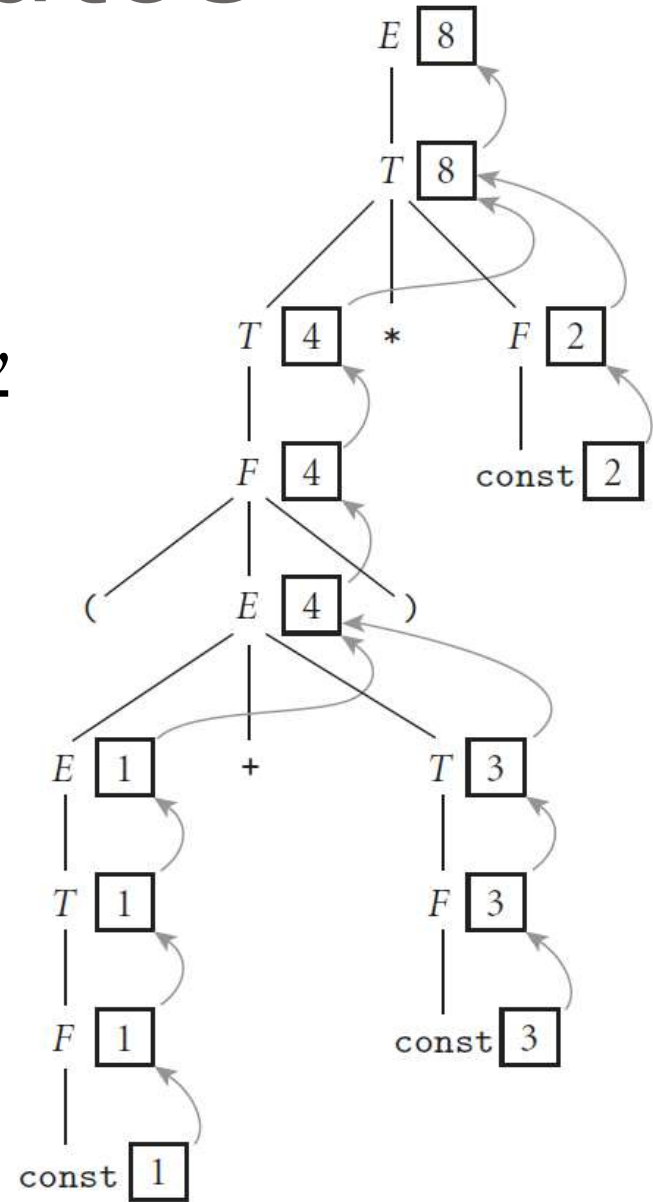
Evaluating Attributes

- The process of evaluating attributes is called *annotation*, or *DECORATION*, of the parse tree
 - When the parse tree under the previous example grammar is fully decorated, the value of the expression will be in the **val** attribute of the root
- The code fragments for the rules are called *SEMANTIC FUNCTIONS*
 - For example:
E1.val = sum(E2.val, T.val)
 - Semantic functions are not allowed to refer to any variables or attributes outside the current production
 - Action routines may do that (see later)

Evaluating Attributes

Decoration of a parse tree for $(1 + 3) * 2$
needs to detect the order of attribute
evaluation:

- Curving arrows show the *attribute flow*
 - Each box holds the output of a single semantic rule
 - The arrow is the input to the rule
- ***synthesized attributes***: their values are calculated (synthesized) only in productions in which their symbol appears on the left-hand side.
- A ***S-attributed grammar*** is a grammar where all attributes are synthesized.



Evaluating Attributes

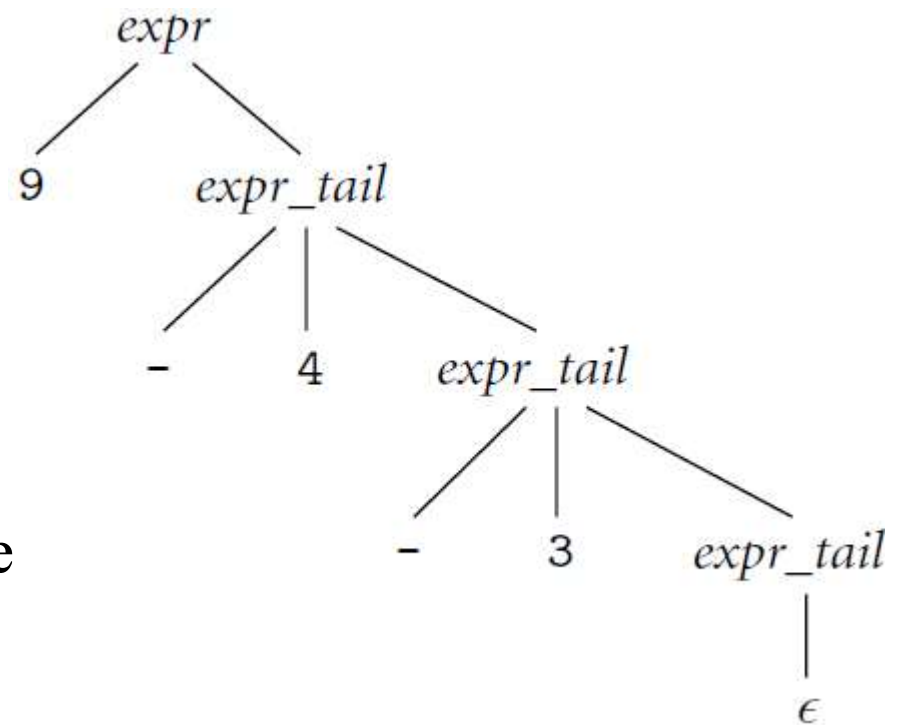
- Tokens have only synthesized attributes, initialized by the scanner (name of an identifier, value of a constant, etc.).
- **INHERITED attributes** may depend on things above or to the side of them in the parse tree:

$expr \longrightarrow const\ expr_tail$

$expr_tail \longrightarrow -\ const\ expr_tail \mid \epsilon$

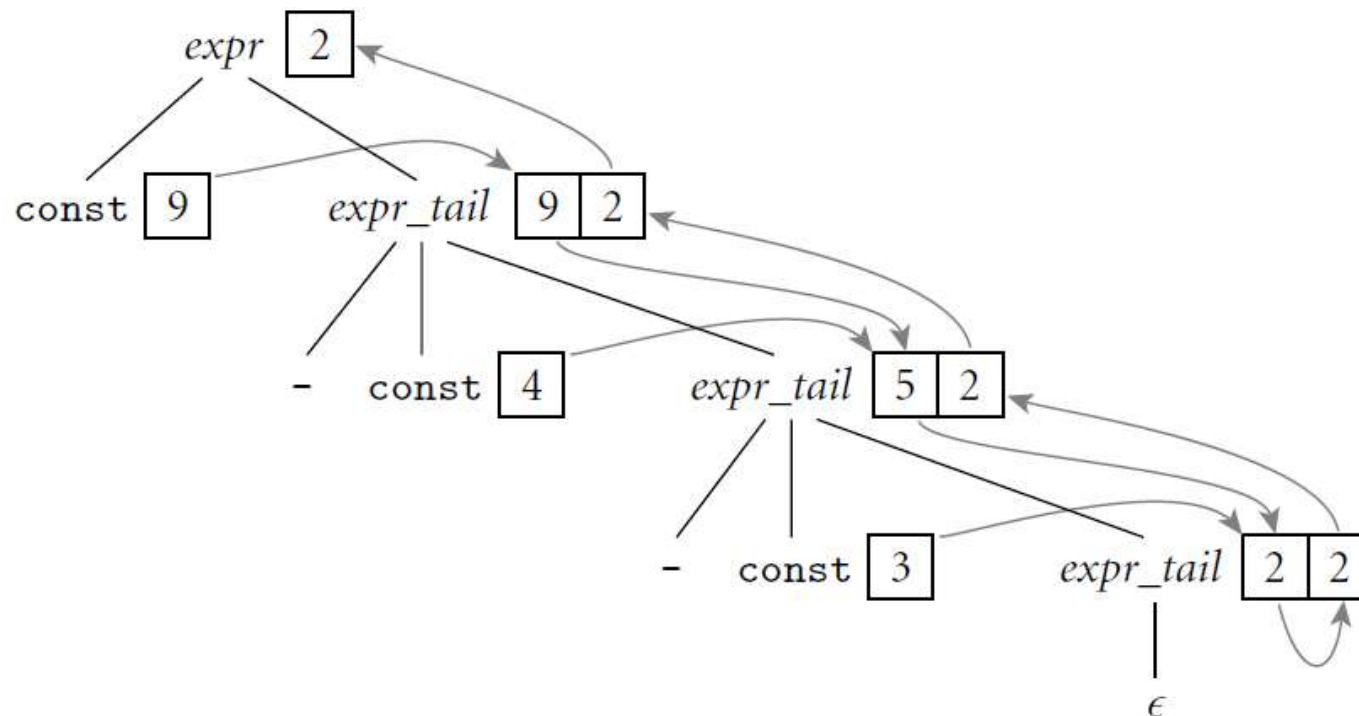
we cannot summarize the right subtree of the root with a single numeric value

subtraction is left associative:
requires us to embed the entire tree into the attributes of a single node



Evaluating Attributes

- Decoration with *left-to-right attribute flow*: pass attribute values not only **bottom-up** but **also left-to-right** in the tree
- 9 can be combined in left-associative fashion with the 4 and
- 5 can then be passed into the middle *expr_tail* node, combined with the 3 to make 2, and then passed upward to the root



Evaluating Attributes

$expr \longrightarrow \text{const } expr_tail$

▷ $expr_tail.st := \text{const.val}$ (1)

▷ $expr.val := expr_tail.val$ (2)

$expr_tail_1 \longrightarrow - \text{const } expr_tail_2$

▷ $expr_tail_2.st := expr_tail_1.st - \text{const.val}$

▷ $expr_tail_1.val := expr_tail_2.val$

$expr_tail \longrightarrow \epsilon$

▷ $expr_tail.val := expr_tail.st$

(1) serves to copy the left context (value of the expression so far) into a “subtotal” (st) attribute.

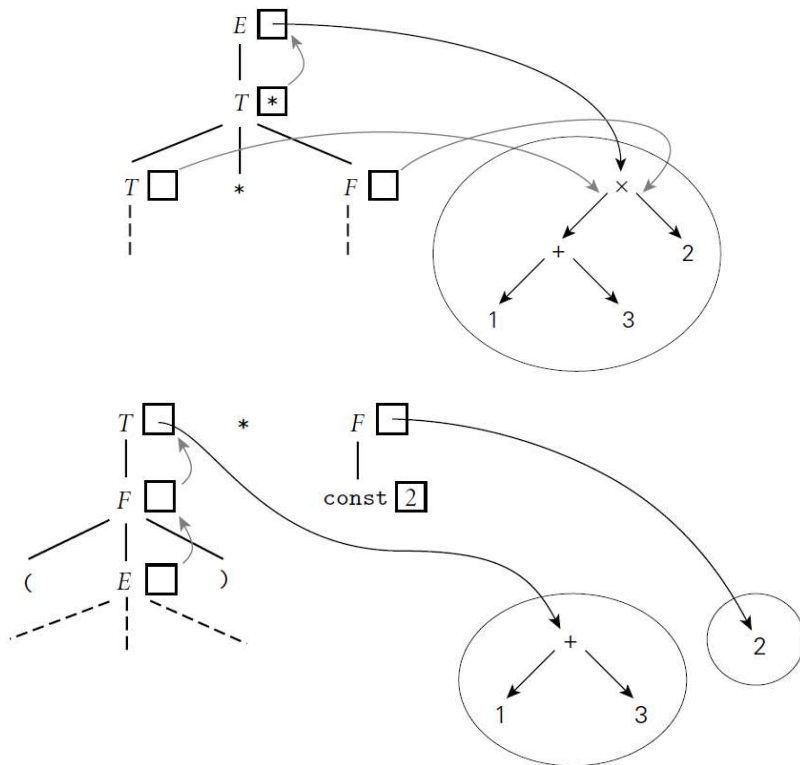
Root rule (2) copies the final value from the right-most leaf back up to the root.

Syntax trees

- A *one-pass compiler* is a compiler that interleaves semantic analysis and code generation with parsing
- *Syntax trees*: if the parsing and code generation are **not interleaved**, then attribute rules must be added to create the syntax tree:
 - The attributes in these grammars point to nodes of the syntax tree (containing unary or binary operators, pointers to the supplied operand(s), etc.)
 - The attributes hold neither numeric values nor target code fragments

Syntax trees

- Bottom-up (S-attributed) attribute grammar to construct a syntax tree



$$E_1 \longrightarrow E_2 + T$$

▷ $E_1.\text{ptr} := \text{make_bin_op}("+", E_2.\text{ptr}, T.\text{ptr})$

$$E_1 \longrightarrow E_2 - T$$

▷ $E_1.\text{ptr} := \text{make_bin_op}("-", E_2.\text{ptr}, T.\text{ptr})$

$$E \longrightarrow T$$

▷ $E.\text{ptr} := T.\text{ptr}$

$$T_1 \longrightarrow T_2 * F$$

▷ $T_1.\text{ptr} := \text{make_bin_op}("x", T_2.\text{ptr}, F.\text{ptr})$

$$T_1 \longrightarrow T_2 / F$$

▷ $T_1.\text{ptr} := \text{make_bin_op}("/", T_2.\text{ptr}, F.\text{ptr})$

$$T \longrightarrow F$$

▷ $T.\text{ptr} := F.\text{ptr}$

$$F_1 \longrightarrow - F_2$$

▷ $F_1.\text{ptr} := \text{make_un_op}("+/-", F_2.\text{ptr})$

$$F \longrightarrow (E)$$

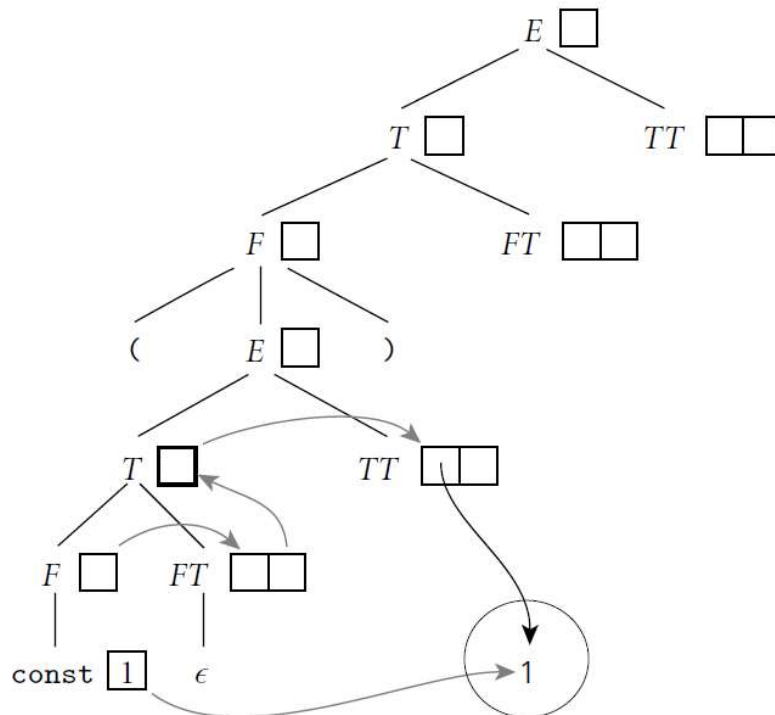
▷ $F.\text{ptr} := E.\text{ptr}$

$$F \longrightarrow \text{const}$$

▷ $F.\text{ptr} := \text{make_leaf}(\text{const.val})$

Syntax trees

- Top-down (L-attributed) attribute grammar to construct a syntax tree:



$$E \rightarrow T TT$$

- $\triangleright TT.st := T.ptr$
- $\triangleright E.ptr := TT.ptr$

$$TT_1 \rightarrow + T TT_2$$

- $\triangleright TT_2.st := \text{make_bin_op}("+", TT_1.st, T.ptr)$
- $\triangleright TT_1.ptr := TT_2.ptr$

$$TT_1 \rightarrow - T TT_2$$

- $\triangleright TT_2.st := \text{make_bin_op}("-", TT_1.st, T.ptr)$
- $\triangleright TT_1.ptr := TT_2.ptr$

$$TT \rightarrow \epsilon$$

- $\triangleright TT.ptr := TT.st$

$$T \rightarrow F FT$$

- $\triangleright FT.st := F.ptr$
- $\triangleright T.ptr := FT.ptr$

$$FT_1 \rightarrow * F FT_2$$

- $\triangleright FT_2.st := \text{make_bin_op}("x", FT_1.st, F.ptr)$
- $\triangleright FT_1.ptr := FT_2.ptr$

$$FT_1 \rightarrow / F FT_2$$

- $\triangleright FT_2.st := \text{make_bin_op}("/\div", FT_1.st, F.ptr)$
- $\triangleright FT_1.ptr := FT_2.ptr$

$$FT \rightarrow \epsilon$$

- $\triangleright FT.ptr := FT.st$

$$F_1 \rightarrow - F_2$$

- $\triangleright F_1.ptr := \text{make_un_op}("+/-", F_2.ptr)$

$$F \rightarrow (E)$$

- $\triangleright F.ptr := E.ptr$

$$F \rightarrow \text{const}$$

- $\triangleright F.ptr := \text{make_leaf}(\text{const.val})$

Action Routines

- While it is possible to construct automatic tools to analyze attribute flow and decorate parse trees, most compilers rely on *action routines*, which the compiler writer embeds in the right-hand sides of productions to evaluate attribute rules at specific points in a parse
- An *action routine* is like a "semantic function" that we tell the compiler to execute at a particular point in the parse
 - In an LL-family parser, action routines can be embedded at arbitrary points in a production's right-hand side
 - They will be executed left to right during parsing

Action Routines

- If semantic analysis and code generation are interleaved with parsing, then action routines can be used to perform semantic checks and generate code
 - Later compilation phases can then consist of ad-hoc tree traversal(s), or can use an automatic tool to generate a translation scheme
- If semantic analysis and code generation are broken out as separate phases, then action routines can be used to build a syntax tree

Action Routines

- Entries in the attributes stack are pushed and popped automatically
- The *syntax tree* is produced

program \rightarrow *item*

int_decl : *item* \rightarrow *id item*

read : *item* \rightarrow *id item*

real_decl : *item* \rightarrow *id item*

write : *item* \rightarrow *expr item*

null : *item* \rightarrow ϵ

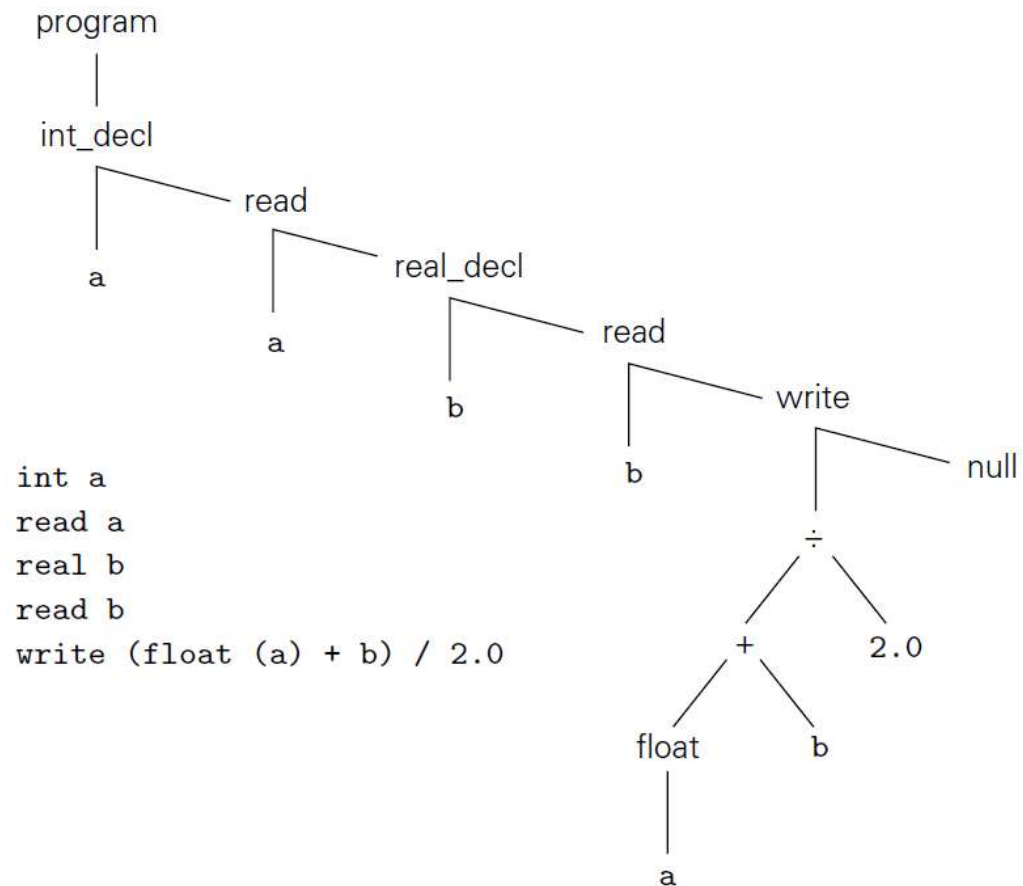
\div : *expr* \rightarrow *expr expr*

$+$: *expr* \rightarrow *expr expr*

float : *expr* \rightarrow *expr*

id : *expr* \rightarrow ϵ

real_const : *expr* \rightarrow ϵ



Decorating a Syntax Tree

- Sample of complete tree grammar representing structure of the syntax tree

```
id : expr  $\rightarrow$   $\epsilon$ 
  ▷ if (id.name, A)  $\in$  expr.symtab      -- for some type A
    expr.errors := null
    expr.type := A
  else
    expr.errors := [id.name "undefined at" id.location]
    expr.type := error

int_const : expr  $\rightarrow$   $\epsilon$ 
  ▷ expr.type := int

real_const : expr  $\rightarrow$   $\epsilon$ 
  ▷ expr.type := real

'+' : expr1  $\rightarrow$  expr2 expr3
  ▷ expr2.symtab := expr1.symtab
  ▷ expr3.symtab := expr1.symtab
  ▷ check_types(expr1, expr2, expr3)

'-' : expr1  $\rightarrow$  expr2 expr3
  ▷ expr2.symtab := expr1.symtab
  ▷ expr3.symtab := expr1.symtab
  ▷ check_types(expr1, expr2, expr3)

'×' : expr1  $\rightarrow$  expr2 expr3
  ▷ expr2.symtab := expr1.symtab
  ▷ expr3.symtab := expr1.symtab
  ▷ check_types(expr1, expr2, expr3)

'÷' : expr1  $\rightarrow$  expr2 expr3
  ▷ expr2.symtab := expr1.symtab
  ▷ expr3.symtab := expr1.symtab
  ▷ check_types(expr1, expr2, expr3)

float : expr1  $\rightarrow$  expr2
  ▷ expr2.symtab := expr1.symtab
  ▷ convert_type(expr2, expr1, int, real, "float of non-int")

trunc : expr1  $\rightarrow$  expr2
  ▷ expr2.symtab := expr1.symtab
  ▷ convert_type(expr2, expr1, real, int, "trunc of non-real")
```