

Semantic Analysis

CSE 307 – Principles of Programming Languages

Stony Brook University

<http://www.cs.stonybrook.edu/~cse307>

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 match the number of formal parameters in the subroutine definition – cannot be counted using CFG, type consistency):
 - 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:
 - 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**.

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 : $i \neq m$

Base Property: $I(0)$ is “ $i = 0$ and $\text{product} = 0 \cdot x = 0$ ”

Inductive Property: [If $G \wedge I(k)$ is true before a loop iteration (where $k \geq 0$), then $I(k+1)$ is true after the loop iteration.]

Let k is a nonnegative integer such that $G \wedge I(k)$ is true

Since $i \neq m$, the guard is passed

$$\text{product} = \text{product} + x = kx + x = (k + 1)x$$

$$i = i + 1 = k + 1$$

$I(k + 1)$: ($i = k + 1$ and $\text{product} = (k + 1)x$) is true

Eventual Falsity of Guard: [After a finite number of iterations of the loop, G becomes false]

After m iterations of the loop: $i = m$ and G becomes false

Correctness of the Post-Condition: [If N is the least number of iterations after which G is false and $I(N)$ is true, then the value of the algorithm variables will be as specified in the post-condition of the loop.]

$I(N)$ is true at the end of the loop: $i = N$ and $\text{product} = N \times x$

G becomes false after N iterations, $i = m$, so $m = i = N$

The post-condition: the value of product after execution of the loop should be $m \times x$ is true.

Static analysis

- *Alias analysis* determines when values can be safely cached in registers, computed “out of order,” or accessed by concurrent threads.
- *Escape analysis* determines when all references to a value will be confined to a given context, allowing it to be allocated on the stack instead of the heap, or to be accessed without locks.
- *Subtype analysis* determines when a variable in an object-oriented language is guaranteed to have a certain subtype, so that its methods can be called without dynamic dispatch.

Other static analysis

- Optimizations:

- *unsafe* if they may lead to incorrect code,
- *speculative* if they usually improve performance, but may degrade it in certain cases
 - *Non-binding prefetches* bring data into the cache before they are needed,
 - *Trace scheduling* rearranges code in hopes of improving the performance of the processor pipeline and the instruction cache.
- A compiler is *conservative* if it applies optimizations only when it can guarantee that they will be both safe and effective.
- A compiler is *optimistic* if it uses speculative optimizations.
 - it may also use unsafe optimizations by generating two versions of the code, with a dynamic check that chooses between them based on information not available at compile time.

Attribute Grammars

- Both semantic analysis and (intermediate) code generation can be described in terms of annotation, or "decoration" of a parse or syntax tree
- ATTRIBUTE GRAMMARS provide a formal framework for decorating such a tree

Attribute Grammars

- LR (bottom-up) grammar for arithmetic expressions made of constants, with precedence and associativity
 - says nothing about what the program MEANS

E	\longrightarrow	$E + T$
E	\longrightarrow	$E - T$
E	\longrightarrow	T
T	\longrightarrow	$T * F$
T	\longrightarrow	T / F
T	\longrightarrow	F
F	\longrightarrow	$- F$
F	\longrightarrow	(E)
F	\longrightarrow	const

Attribute Grammars *semantic function*

- Attributed grammar:
 - define the semantics of the input program
 - Associates expressions to mathematical concepts!!!
 - Attribute rules are definitions, not assignments: they are not necessarily meant to be evaluated at any particular time, or in any particular order

$E_1 \longrightarrow E_2 + T$ (sum, etc.)

▷ $E_1.val := \text{sum}(E_2.val, T.val)$

$E_1 \longrightarrow E_2 - T$

▷ $E_1.val := \text{difference}(E_2.val, T.val)$

$E \longrightarrow T$

▷ $E.val := T.val$ *copy rule*

$T_1 \longrightarrow T_2 * F$

▷ $T_1.val := \text{product}(T_2.val, F.val)$

$T_1 \longrightarrow T_2 / F$

▷ $T_1.val := \text{quotient}(T_2.val, F.val)$

$T \longrightarrow F$

▷ $T.val := F.val$

$F_1 \longrightarrow - F_2$

▷ $F_1.val := \text{additive_inverse}(F_2.val)$

$F \longrightarrow (E)$

▷ $F.val := E.val$

$F \longrightarrow \text{const}$

▷ $F.val := \text{const.val}$

Attribute Grammars Example

Tokens: int (attr val)

var (attr name)

$S \rightarrow \text{var} = E$

$\quad \rightarrow \text{assign}(\text{var.name}, E.\text{val})$

$E1 \rightarrow E2 + T$

$\quad \rightarrow E1.\text{val} = \text{add}(E2.\text{val}, T.\text{val})$

$E1 \rightarrow E2 - T$

$\quad \rightarrow E1.\text{val} = \text{sub}(E2.\text{val}, T.\text{val})$

$E \rightarrow T$

$\quad \rightarrow E.\text{val} = T.\text{val}$

$T \rightarrow \text{var}$

$\quad \rightarrow T.\text{val} = \text{lookup}(\text{var.name})$

$T \rightarrow \text{int}$

$\quad \rightarrow T.\text{val} = \text{int.val}$

“bar = 50

foo = 100 + 200 – bar”

Attribute Grammars

- Attributed grammar to count the elements of a list:

$L \longrightarrow \text{id}$	$\triangleright L_1.C := 1$
$L_1 \longrightarrow L_2, \text{id}$	$\triangleright L_1.C := L_2.C + 1$

- Semantic functions are not allowed to refer to any variables or attributes outside the current production
- Action routines may do that (see later).

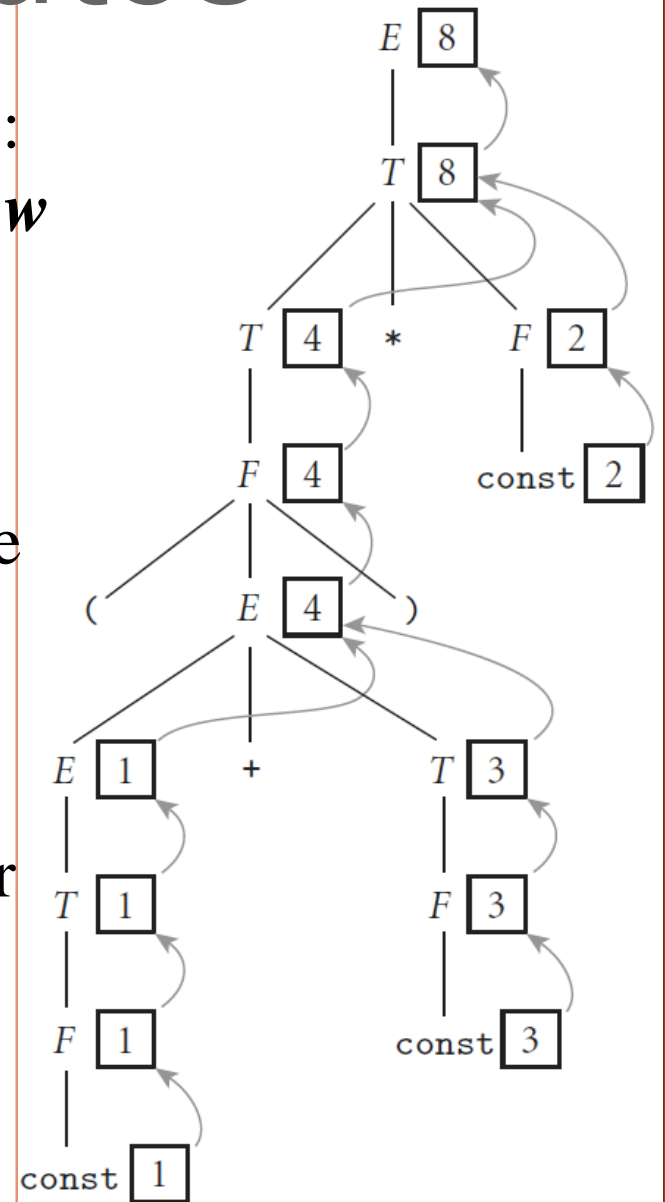
Evaluating Attributes

- The process of evaluating attributes is called *annotation*, or **DECORATION**, of the parse tree
 - When a parse tree under this 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**
 - Strictly speaking, they should be cast as functions, e.g.,
$$E1.val = \text{sum}(E2.val, T.val)$$

Evaluating Attributes

Decoration of a parse tree for $(1 + 3) * 2$:

- 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.



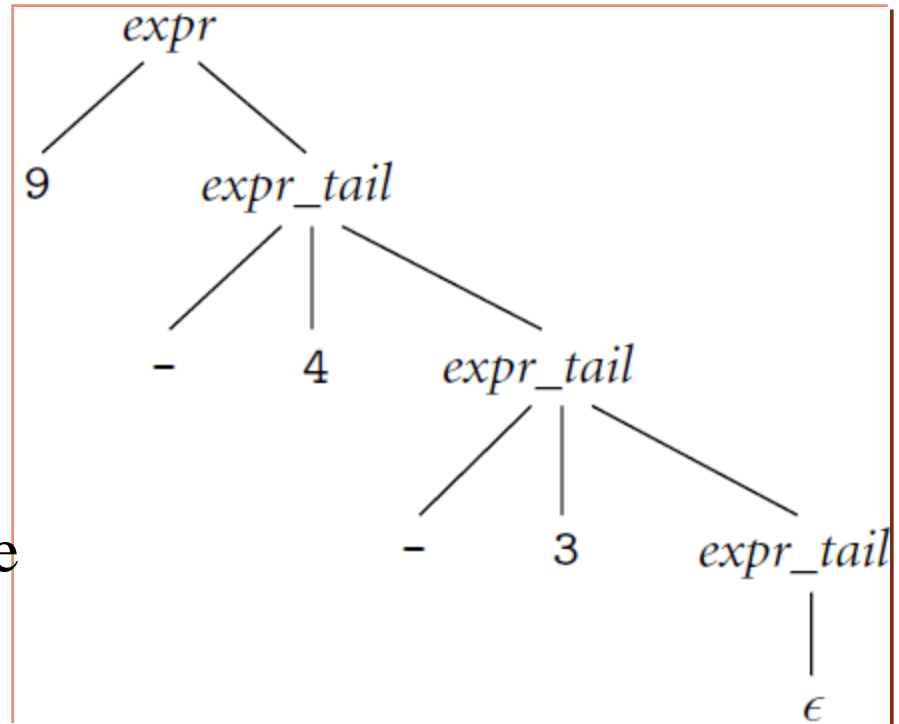
Evaluating Attributes

- 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, e.g., LL(1) grammar:

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

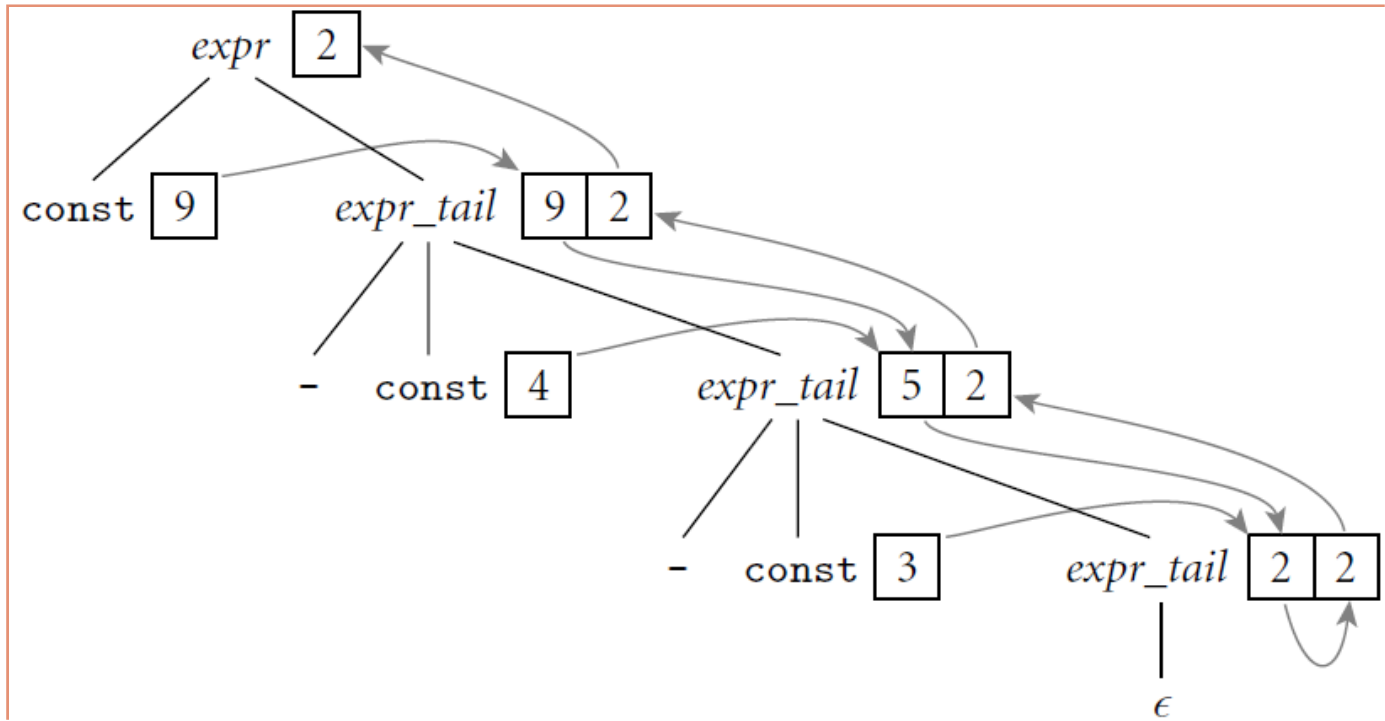
we cannot summarize the right subtree of the root with a single numeric value

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.
- 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.

Evaluating Attributes

An attribute grammar for constant expressions based on an LL(1) CFG

- An attribute grammar is **well defined** if its rules determine a unique set of values for the attributes of every possible parse tree.
- An attribute grammar is **noncircular** if it never leads to a parse tree in which there are cycles in the attribute flow graph.

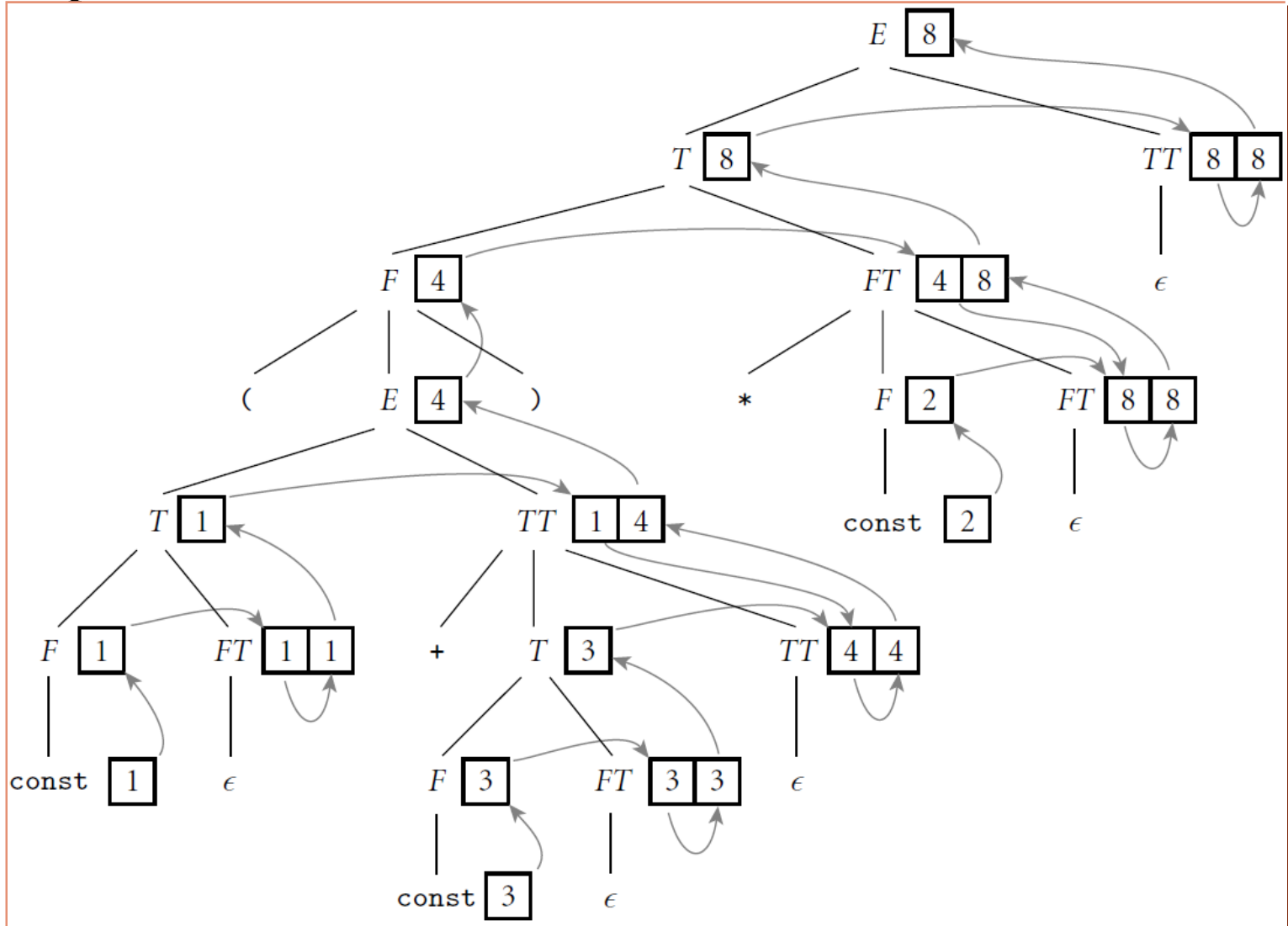
1. $E \rightarrow T TT$
▷ $TT.st := T.val$ ▷ $E.val := TT.val$
2. $TT_1 \rightarrow + T TT_2$
▷ $TT_2.st := TT_1.st + T.val$ ▷ $TT_1.val := TT_2.val$
3. $TT_1 \rightarrow - T TT_2$
▷ $TT_2.st := TT_1.st - T.val$ ▷ $TT_1.val := TT_2.val$
4. $TT \rightarrow \epsilon$
▷ $TT.val := TT.st$
5. $T \rightarrow F FT$
▷ $FT.st := F.val$ ▷ $T.val := FT.val$
6. $FT_1 \rightarrow * F FT_2$
▷ $FT_2.st := FT_1.st \times F.val$ ▷ $FT_1.val := FT_2.val$
7. $FT_1 \rightarrow / F FT_2$
▷ $FT_2.st := FT_1.st \div F.val$ ▷ $FT_1.val := FT_2.val$
8. $FT \rightarrow \epsilon$
▷ $FT.val := FT.st$
9. $F_1 \rightarrow - F_2$
▷ $F_1.val := - F_2.val$
10. $F \rightarrow (E)$
▷ $F.val := E.val$
11. $F \rightarrow \text{const}$
▷ $F.val := \text{const.val}$

Evaluating Attributes

- Synthesized Attributes (S-attributed grammar):
 - Data flows bottom-up,
 - From RHS to LHS only,
 - Can be parsed by LR grammar.
- Inherited Attributes:
 - Data flows top-down and bottom-up.
 - Can be parsed with LL grammar.

Evaluating Attributes

top-down parse tree for $(1 + 3) * 2$



Evaluating Attributes

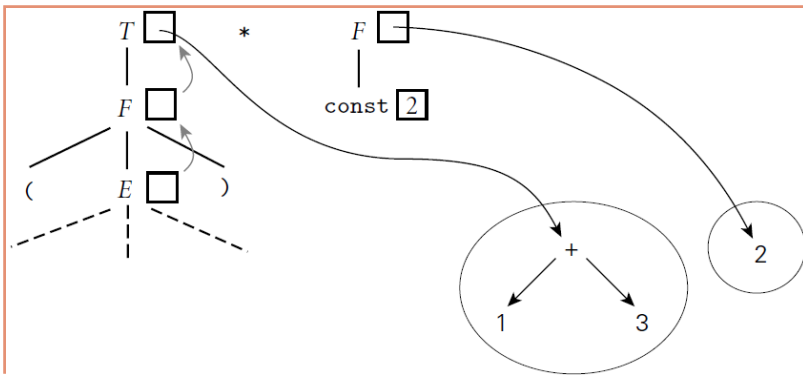
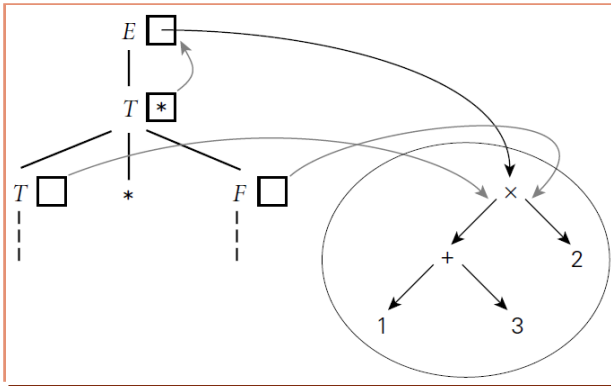
- A *translation scheme* is an algorithm that decorates parse trees by invoking the rules of an attribute grammar in an order consistent with the tree's attribute flow.
- An *oblivious* scheme makes repeated passes over a tree, invoking any semantic function whose arguments have all been defined, and stopping when it completes a pass in which no values change.
- A *dynamic* scheme that tailors the evaluation order to the structure of the given parse tree, e.g., by constructing a topological sort of the attribute flow graph and then invoking rules in an order consistent with the sort.
- An attribute grammar is *L-attributed* if its attributes can be evaluated by visiting the nodes of the parse tree in a single left-to-right, depth-first traversal (same order with a top-down parse).

Syntax trees

- A *one-pass compiler* is a compiler that interleaves semantic analysis and code generation with parsing.
- *Syntax trees*: if the parsing and semantic analysis 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



$$E_1 \longrightarrow E_2 + T$$

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

$$E_1 \longrightarrow E_2 - T$$

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

$$E \longrightarrow T$$

$$\triangleright E.\text{ptr} := T.\text{ptr}$$

$$T_1 \longrightarrow T_2 * F$$

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

$$T_1 \longrightarrow T_2 / F$$

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

$$T \longrightarrow F$$

$$\triangleright T.\text{ptr} := F.\text{ptr}$$

$$F_1 \longrightarrow - F_2$$

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

$$F \longrightarrow (E)$$

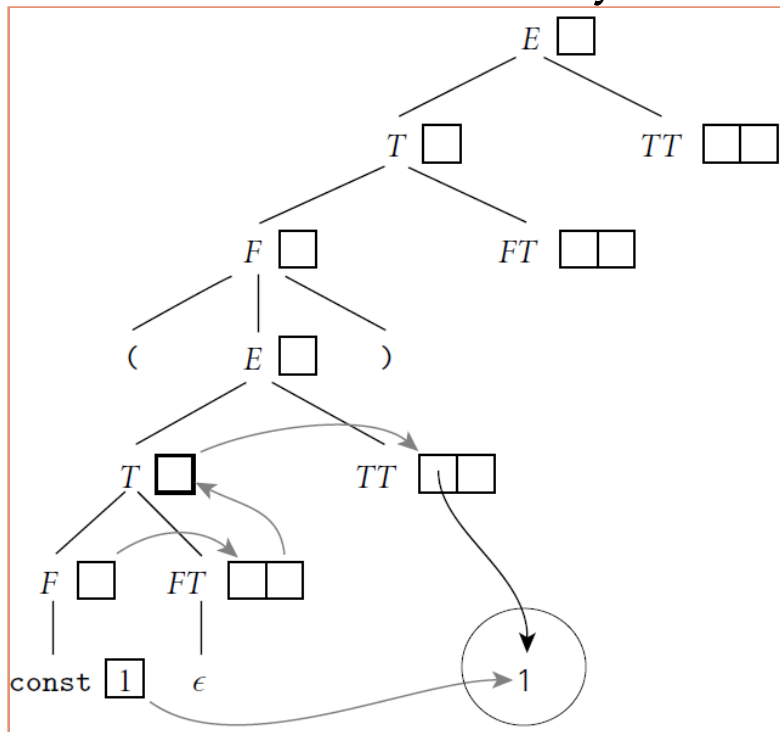
$$\triangleright F.\text{ptr} := E.\text{ptr}$$

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

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

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

- If we break semantic analysis and code generation out into separate phase(s), then the code that builds the parse/syntax tree must still use a left-to-right (L-attributed) translation scheme
- However, the later phases are free to use a fancier translation scheme if they want
- There are automatic tools that generate translation schemes for context-free grammars or tree grammars (which describe the possible structure of a syntax tree)
 - These tools are heavily used in syntax-based editors and incremental compilers
 - **Most ordinary compilers, however, use ad-hoc techniques:**
action routines

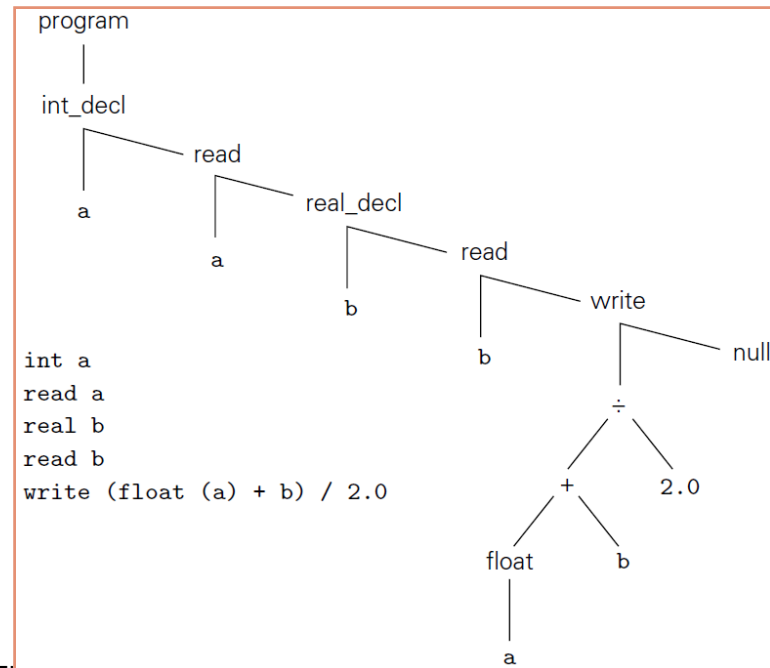
Action Routines

- An ad-hoc translation scheme that is interleaved with parsing takes the form of a set of ACTION ROUTINES:
 - An *action routine* is a semantic function that we tell the compiler to execute at a particular point in the parse
- If semantic analysis and code generation are interleaved with parsing, then action routines can be used to perform semantic checks and generate code.
- 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

- Later compilation phases can then consist of ad-hoc tree traversal(s), or can use an automatic tool to generate a translation scheme
 - 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  $\rightarrow$   $\epsilon$   
'÷' : expr  $\rightarrow$  expr expr  
'+' : expr  $\rightarrow$  expr expr  
float : expr  $\rightarrow$  expr  
id : expr  $\rightarrow$   $\epsilon$   
real_const : expr  $\rightarrow$   $\epsilon$ 
```



Decorating a Syntax Tree

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

```
id : expr  $\longrightarrow$   $\epsilon$ 
  ▷ 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  $\longrightarrow$   $\epsilon$ 
  ▷ expr.type := int

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

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

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

'x' : expr1  $\longrightarrow$  expr2 expr3
  ▷ expr2.symtab := expr1.symtab
  ▷ expr3.symtab := expr1.symtab
  ▷ check_types(expr1, expr2, expr3)

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

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

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