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 matches 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

- 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)
- Following parsing, the next two phases of the "typical" compiler are:
  - semantic analysis
  - (intermediate) code generation
- The principal job of the ***semantic analyzer*** is to enforce <u>static semantic rules</u>, plus:
  - constructs a syntax tree
  - information gathered is needed by the code generator

Attribute grammars

- However, **parsing, semantic analysis, and intermediate code generation are typically interleaved**:
  - a common approach interleaves parsing construction of a syntax tree with phases for semantic analysis and code generation
    - replaces the parse tree with a syntax tree that reflects the input program in a more straightforward way
  - The semantic analysis and intermediate code generation ***annotate*** the parse tree with ***attributes***
    - These kind of grammars are called ***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

Dynamic checks

- 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*"

Dynamic checks

- ***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 syntax: **`assert denominator != 0;`**
    - An **`AssertionError`** exception will be thrown if the semantic check fails at run time.
  - C syntax: **`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 with the message as a parameter

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 <u>default</u>, the assertions are <u>disabled</u> 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.
  - <u>Assertions are to assure the correctness of the program</u>
- 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 vs. Exceptions which cannot be turned on or off

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!
  - A common use of assertions is to represent assumptions with assertions in the code
  - This gives you more confidence to assure correctness of the program
  - 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;
         // this default case should never be reached
}
```

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, the 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 it 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 product $= 0$]

while ($i \neq m$)

product := product + x

i := i + 1

end while

[Post-condition: product = mx]

Loop invariant I(n): $i = n$ and product $= n*x$

Guard G: $i \neq m$

Base Property: I (0) is "$i = 0$ and 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

product = product + x = kx + x = (k + 1)x

$i = i + 1 = k + 1$

$I(k + 1)$: ($i = k + 1$ and 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 product = Nx

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*x is true.

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

- Examples of 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

- Static analysis is usually done for **<u>Optimizations</u>**:
  - optimizations can be ***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
- 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
- The attributes are divided into two groups: ***synthesized*** attributes and ***inherited*** attributes
  - ***Synthesized***: the value is computed from the values of attributes of the children
    - ***S-attributed grammar*** = synthesized attributes only

Attribute Grammars

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

$$
\begin{aligned}
E &\longrightarrow E + T \\
E &\longrightarrow E - T \\
E &\longrightarrow T \\
T &\longrightarrow T * F \\
T &\longrightarrow T \,/\, F \\
T &\longrightarrow F \\
F &\longrightarrow - F \\
F &\longrightarrow (E) \\
F &\longrightarrow \texttt{const}
\end{aligned}
$$

Attribute Grammars

- **Attributed grammar**:
  - defines 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

semantic function (sum, etc.)

$E_1 \longrightarrow E_2 + T$
$\quad \triangleright$ E$_1$.val := sum(E$_2$.val, T.val)

$E_1 \longrightarrow E_2 - T$
$\quad \triangleright$ E$_1$.val := difference(E$_2$.val, T.val)

$E \longrightarrow T$
$\quad \triangleright$ E.val := T.val *copy rule*

$T_1 \longrightarrow T_2 * F$
$\quad \triangleright$ T$_1$.val := product(T$_2$.val, F.val)

$T_1 \longrightarrow T_2 / F$
$\quad \triangleright$ T$_1$.val := quotient(T$_2$.val, F.val)

$T \longrightarrow F$
$\quad \triangleright$ T.val := F.val

$F_1 \longrightarrow - F_2$
$\quad \triangleright$ F$_1$.val := additive_inverse(F$_2$.val)

$F \longrightarrow (E)$
$\quad \triangleright$ F.val := E.val

$F \longrightarrow$ const
$\quad \triangleright$ F.val := const.val

Attribute Grammars

- Attributed grammar to count the elements of a list:

$$L \longrightarrow \texttt{id} \qquad \triangleright \quad L_1.c := 1$$

$$L_1 \longrightarrow L_2 \texttt{ , id} \qquad \triangleright \quad L_1.c := L_2.c + 1$$

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
```

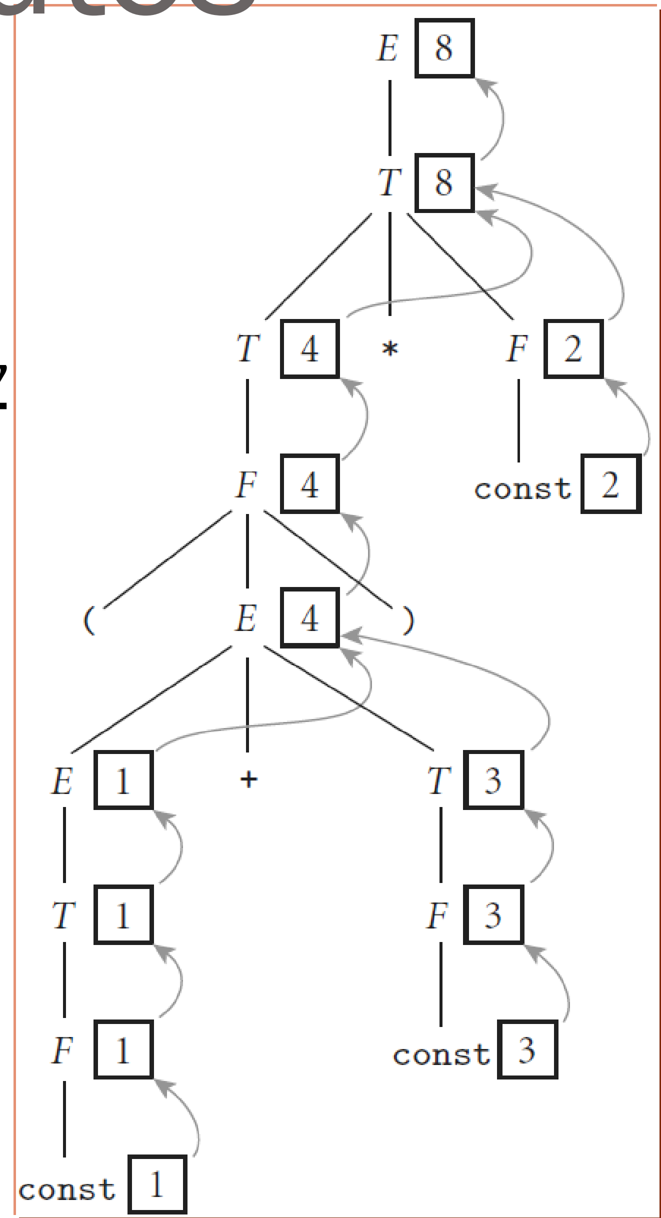
Input:
"bar = 50
foo = 100 + 200 – bar"

Evaluating Attributes

- 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.

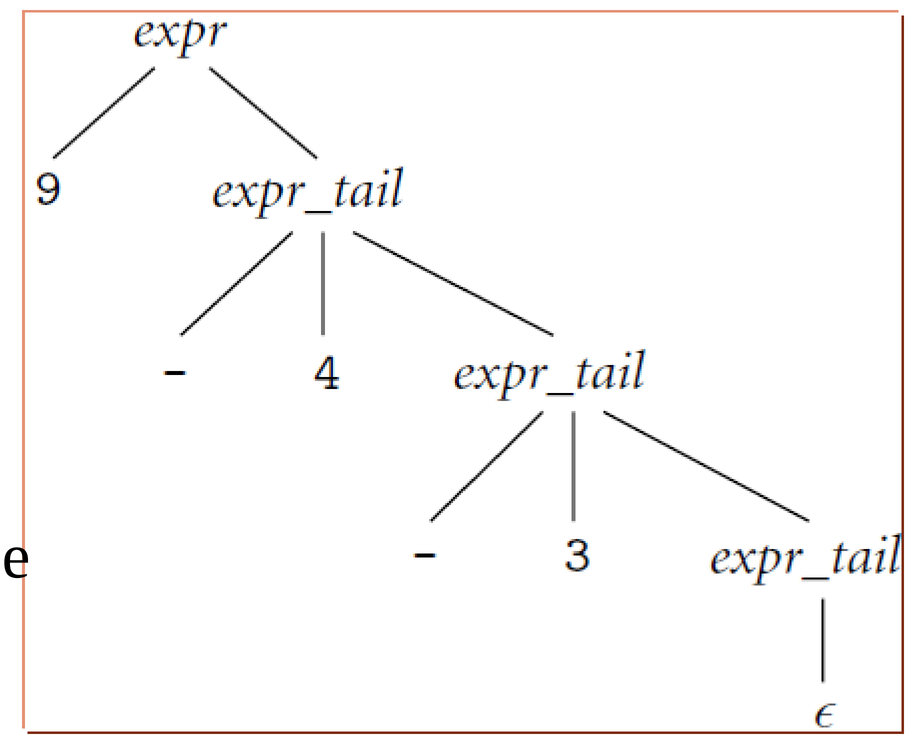
Evaluating Attributes

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

$$expr \longrightarrow \texttt{const}\ expr\_tail$$
$$expr\_tail \longrightarrow \texttt{-}\ \texttt{const}\ expr\_tail \mid \epsilon$$

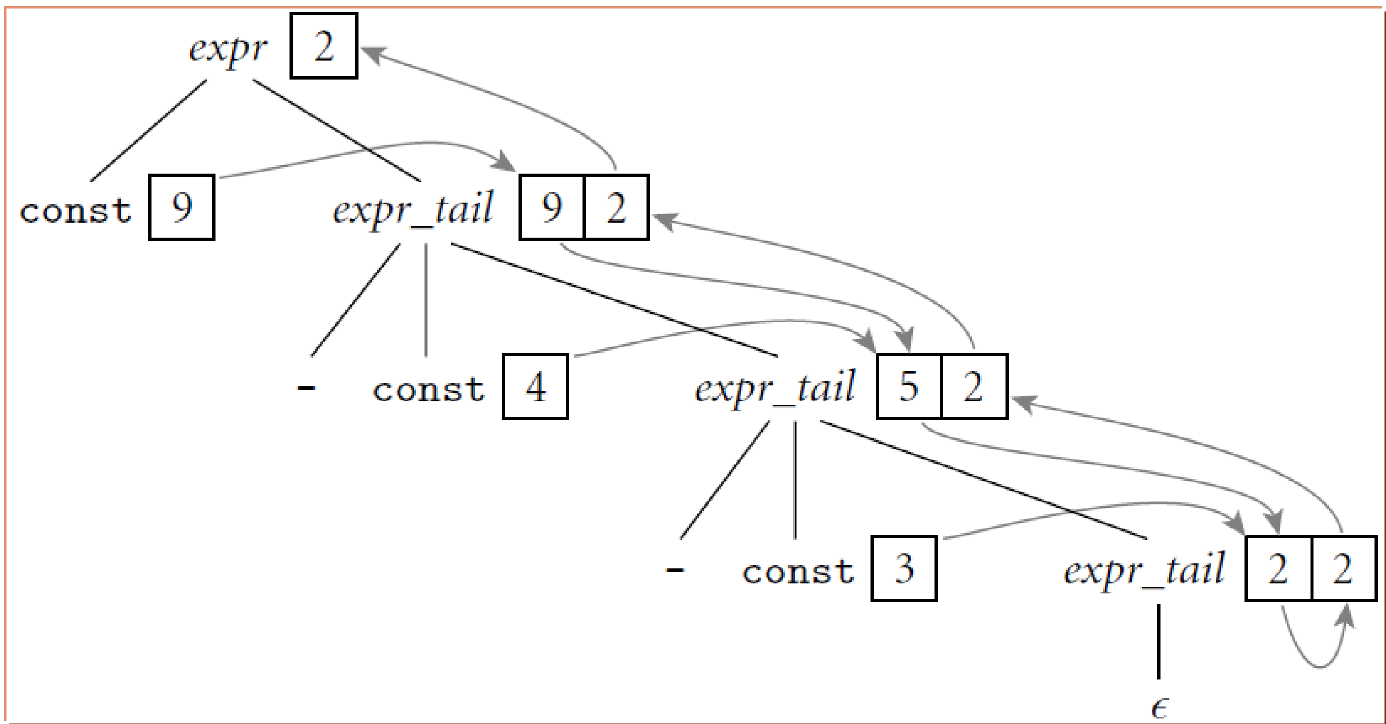
we cannot summarize the right subtree of the root with a single numeric value

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

$expr \longrightarrow \texttt{const}\ expr\_tail$

$\triangleright$ expr_tail.st := const.val (1)

$\triangleright$ expr.val := expr_tail.val (2)

$expr\_tail_1 \longrightarrow \texttt{-}\ \texttt{const}\ expr\_tail_2$

$\triangleright$ $\text{expr\_tail}_2\text{.st} := \text{expr\_tail}_1\text{.st} - \text{const.val}$

$\triangleright$ $\text{expr\_tail}_1\text{.val} := \text{expr\_tail}_2\text{.val}$

$expr\_tail \longrightarrow \epsilon$

$\triangleright$ 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

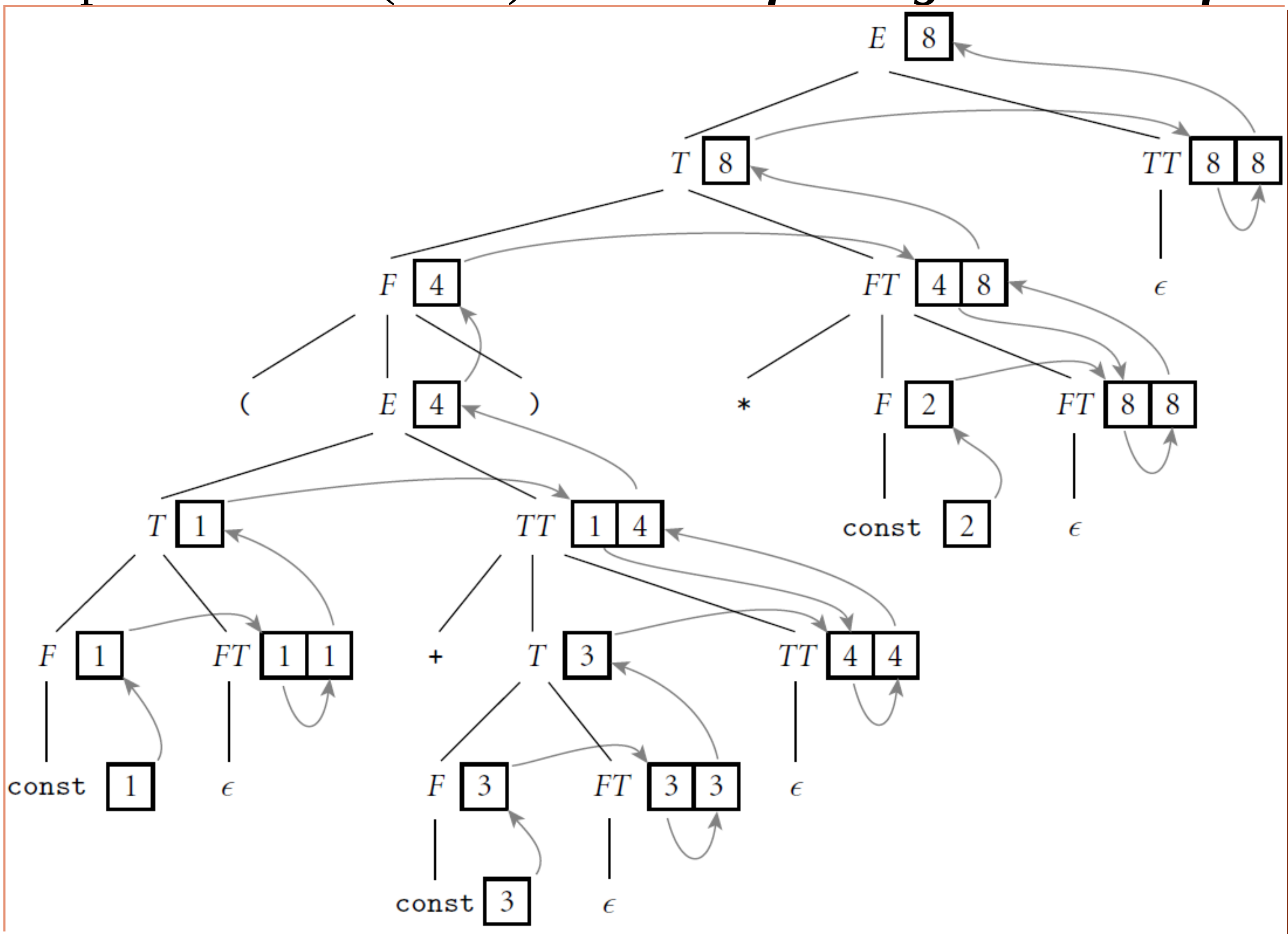
An attribute grammar for constant expressions based on an LL(1) CFG

- 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. $E \longrightarrow T\ TT$
   - $\triangleright$ TT.st := T.val $\quad\triangleright$ E.val := TT.val
2. $TT_1 \longrightarrow$ `+` $T\ TT_2$
   - $\triangleright$ TT_2.st := TT_1.st + T.val $\quad\triangleright$ TT_1.val := TT_2.val
3. $TT_1 \longrightarrow$ `-` $T\ TT_2$
   - $\triangleright$ TT_2.st := TT_1.st − T.val $\quad\triangleright$ TT_1.val := TT_2.val
4. $TT \longrightarrow \epsilon$
   - $\triangleright$ TT.val := TT.st
5. $T \longrightarrow F\ FT$
   - $\triangleright$ FT.st := F.val $\quad\triangleright$ T.val := FT.val
6. $FT_1 \longrightarrow$ `*` $F\ FT_2$
   - $\triangleright$ FT_2.st := FT_1.st × F.val $\quad\triangleright$ FT_1.val := FT_2.val
7. $FT_1 \longrightarrow$ `/` $F\ FT_2$
   - $\triangleright$ FT_2.st := FT_1.st ÷ F.val $\quad\triangleright$ FT_1.val := FT_2.val
8. $FT \longrightarrow \epsilon$
   - $\triangleright$ FT.val := FT.st
9. $F_1 \longrightarrow$ `-` F_2
   - $\triangleright$ F_1.val := − F_2.val
10. $F \longrightarrow$ `(` E `)`
    - $\triangleright$ F.val := E.val
11. $F \longrightarrow$ `const`
    - $\triangleright$ F.val := const.val

Evaluating Attributes

Top-down parse tree for (1 + 3) * 2 with ***left-to-right attribute flow***



Evaluating Attributes

- ***Synthesized Attributes*** (***S-attributed grammars***):
  - 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

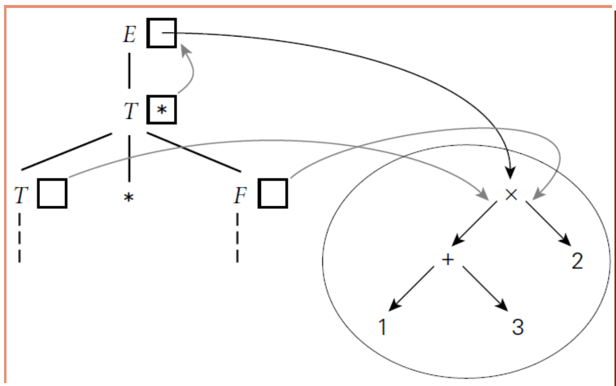
- A ***translation scheme*** is an algorithm that decorates parse trees by invoking the rules of an attribute grammar in an order consistent with the tree's attribute flow
  - An ***oblivious*** scheme makes repeated passes over a tree, invoking any semantic function whose arguments have all been defined, 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)

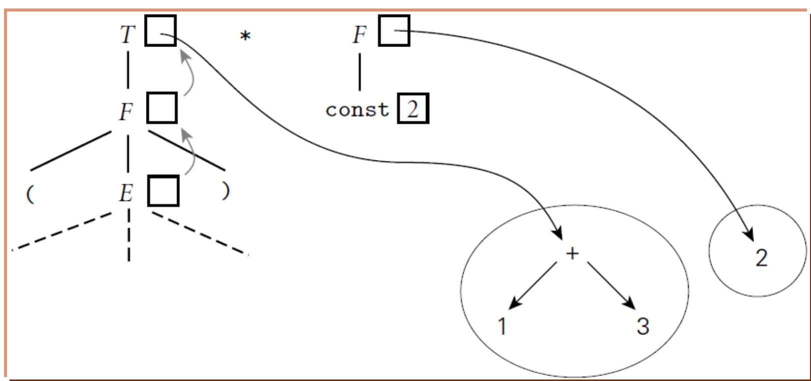
Syntax trees

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





$E_1 \longrightarrow E_2 + T$
$\triangleright$ E₁.ptr := make_bin_op("+", E₂.ptr, T.ptr)

$E_1 \longrightarrow E_2 - T$
$\triangleright$ E₁.ptr := make_bin_op("−", E₂.ptr, T.ptr)

$E \longrightarrow T$
$\triangleright$ E.ptr := T.ptr

$T_1 \longrightarrow T_2 * F$
$\triangleright$ T₁.ptr := make_bin_op("×", T₂.ptr, F.ptr)

$T_1 \longrightarrow T_2 / F$
$\triangleright$ T₁.ptr := make_bin_op("÷", T₂.ptr, F.ptr)

$T \longrightarrow F$
$\triangleright$ T.ptr := F.ptr

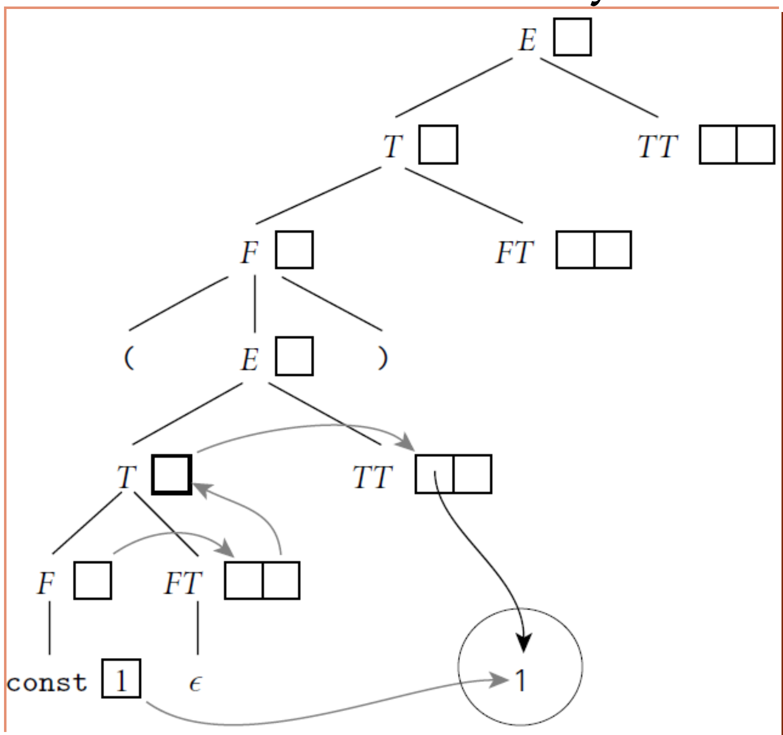
$F_1 \longrightarrow - F_2$
$\triangleright$ F₁.ptr := make_un_op("+/_", F₂.ptr)

$F \longrightarrow (E)$
$\triangleright$ F.ptr := E.ptr

$F \longrightarrow$ const
$\triangleright$ F.ptr := make_leaf(const.val)

Syntax trees

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



$E \longrightarrow T\ TT$
▷ TT.st := T.ptr
▷ E.ptr := TT.ptr

$TT_1 \longrightarrow$ + $T\ TT_2$
▷ TT_2.st := make_bin_op("+", TT_1.st, T.ptr)
▷ TT_1.ptr := TT_2.ptr

$TT_1 \longrightarrow$ - $T\ TT_2$
▷ TT_2.st := make_bin_op("−", TT_1.st, T.ptr)
▷ TT_1.ptr := TT_2.ptr

$TT \longrightarrow \epsilon$
▷ TT.ptr := TT.st

$T \longrightarrow F\ FT$
▷ FT.st := F.ptr
▷ T.ptr := FT.ptr

$FT_1 \longrightarrow$ * $F\ FT_2$
▷ FT_2.st := make_bin_op("×", FT_1.st, F.ptr)
▷ FT_1.ptr := FT_2.ptr

$FT_1 \longrightarrow$ / $F\ FT_2$
▷ FT_2.st := make_bin_op("÷", FT_1.st, F.ptr)
▷ FT_1.ptr := FT_2.ptr

$FT \longrightarrow \epsilon$
▷ FT.ptr := FT.st

$F_1 \longrightarrow$ - F_2
▷ F_1.ptr := make_un_op("+/_", F_2.ptr)

$F \longrightarrow$ (E)
▷ F.ptr := E.ptr

$F \longrightarrow$ const
▷ F.ptr := make_leaf(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

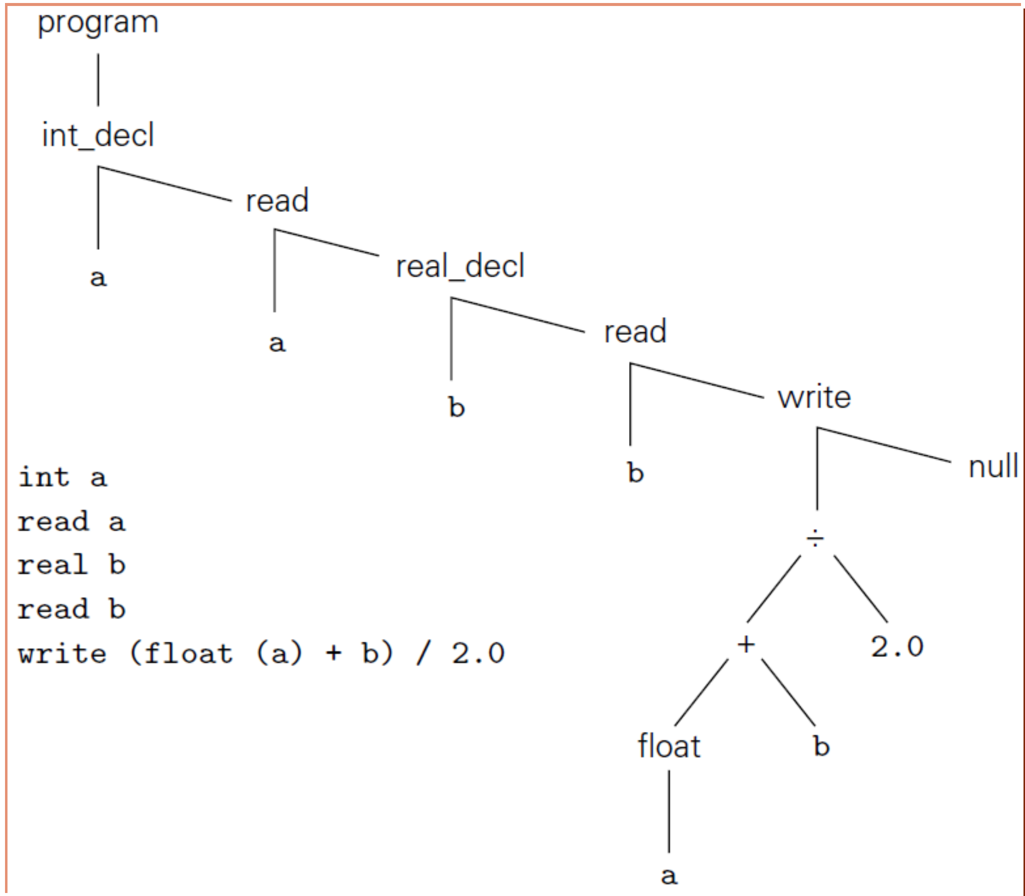
Action Routines

- 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

$$
\begin{array}{l}
\mathit{program} \longrightarrow \mathit{item} \\
\mathit{int\_decl} : \mathit{item} \longrightarrow \mathit{id}\ \mathit{item} \\
\mathit{read} : \mathit{item} \longrightarrow \mathit{id}\ \mathit{item} \\
\mathit{real\_decl} : \mathit{item} \longrightarrow \mathit{id}\ \mathit{item} \\
\mathit{write} : \mathit{item} \longrightarrow \mathit{expr}\ \mathit{item} \\
\\
\mathit{null} : \mathit{item} \longrightarrow \epsilon \\
\text{`}\div\text{'} : \mathit{expr} \longrightarrow \mathit{expr}\ \mathit{expr} \\
\text{`}+\text{'} : \mathit{expr} \longrightarrow \mathit{expr}\ \mathit{expr} \\
\mathit{float} : \mathit{expr} \longrightarrow \mathit{expr} \\
\mathit{id} : \mathit{expr} \longrightarrow \epsilon \\
\mathit{real\_const} : \mathit{expr} \longrightarrow \epsilon
\end{array}
$$




```
int a
read a
real b
read b
write (float (a) + b) / 2.0
```

Decorating a Syntax Tree

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

id : *expr* $\longrightarrow$ ϵ
 ▷ if ⟨id.name, A⟩ ∈ 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* $\longrightarrow$ ϵ
 ▷ expr.type := int

real_const : *expr* $\longrightarrow$ ϵ
 ▷ expr.type := real

'+' : $expr_1 \longrightarrow expr_2\ expr_3$
 ▷ $expr_2$.symtab := $expr_1$.symtab
 ▷ $expr_3$.symtab := $expr_1$.symtab
 ▷ check_types($expr_1$, $expr_2$, $expr_3$)

'−' : $expr_1 \longrightarrow expr_2\ expr_3$
 ▷ $expr_2$.symtab := $expr_1$.symtab
 ▷ $expr_3$.symtab := $expr_1$.symtab
 ▷ check_types($expr_1$, $expr_2$, $expr_3$)

'×' : $expr_1 \longrightarrow expr_2\ expr_3$
 ▷ $expr_2$.symtab := $expr_1$.symtab
 ▷ $expr_3$.symtab := $expr_1$.symtab
 ▷ check_types($expr_1$, $expr_2$, $expr_3$)

'÷' : $expr_1 \longrightarrow expr_2\ expr_3$
 ▷ $expr_2$.symtab := $expr_1$.symtab
 ▷ $expr_3$.symtab := $expr_1$.symtab
 ▷ check_types($expr_1$, $expr_2$, $expr_3$)

float : $expr_1 \longrightarrow expr_2$
 ▷ $expr_2$.symtab := $expr_1$.symtab
 ▷ convert_type($expr_2$, $expr_1$, int, real, "float of non-int")

trunc : $expr_1 \longrightarrow expr_2$
 ▷ $expr_2$.symtab := $expr_1$.symtab
 ▷ convert_type($expr_2$, $expr_1$, real, int, "trunc of non-real")