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

- 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`** would be used if the assertion would be **`false`**
  - For the second assert statement with a detail message, an appropriate **`AssertionError`** constructor would be 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 program **correctness**
  - Exception handling addresses ***robustness*** and assertion addresses ***correctness***

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, 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 ≠ 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 ≠ m

Base Property: I (0) is "i = 0 and product = 0· x = 0"

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

Let k is a nonnegative integer such that G ∧ I (k) is true

Since i ≠ 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 **Optimizations**:
  - optimizations can be ***unsafe*** if they may lead to incorrect code
  - ***speculative*** if they usually improve performance, but may degrade it in certain cases, like:
    - ***non-binding prefetches***, which brings data into the cache before they are needed
    - ***trace scheduling***, which 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, like it may generate 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)

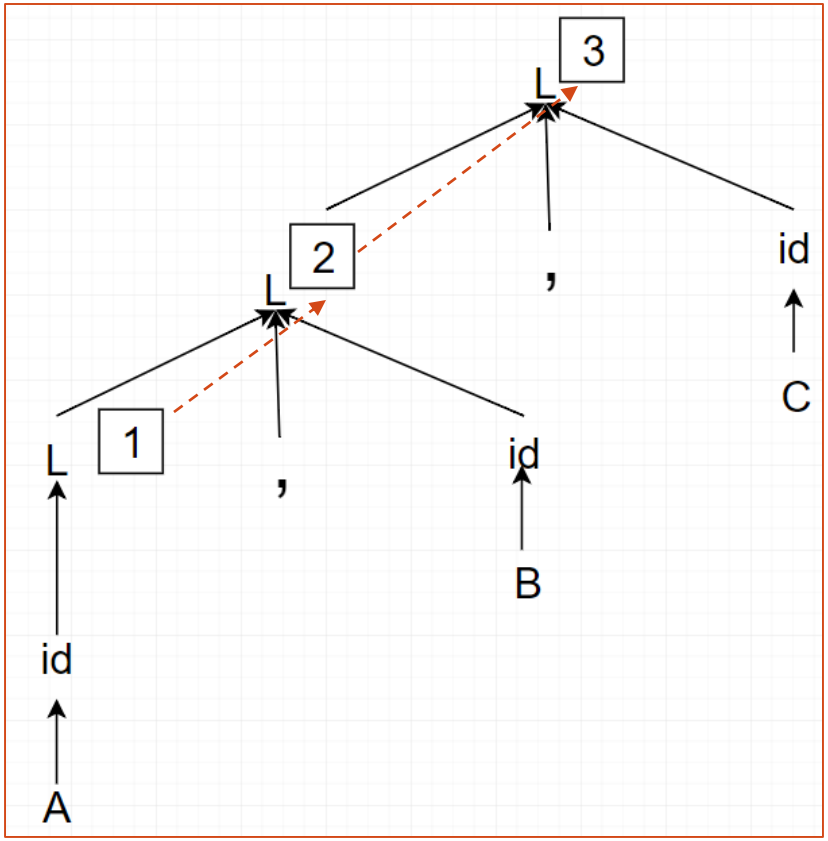
Attribute Grammars

- 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 attribute of a node in the tree is computed from the values of attributes of the children
    - ***S-attributed grammar*** = synthesized attributes only

Attribute Grammars

- Attributed grammar to count the elements of a list:

$L \longrightarrow$ id	▷ $L_1.c := 1$
$L_1 \longrightarrow L_2$, id	▷ $L_1.c := L_2.c + 1$



Ply in python

```
# Tokens
tokens = (
    'ID','COMMA'
    )
t_ID = r'[a-z][a-zA-Z0-9]*'
t_COMMA = r','
# Ignored characters
t_ignore = " \t"
def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
import ply.lex as lex
lex.lex()
```

Ply in python

```
# Parsing rules
def p_list1(t):
    'list : ID'
    t[0] = 1
def p_list2(t):
    'list : list COMMA ID'
    t[0] = t[1] + 1

def p_error(t):
    print("Syntax error at '%s'" % t.value)

import ply.yacc as yacc
yacc.yacc()
try:
    s = "a,b,c"
    a = yacc.parse(s)
    print(a)
except EOFError as e:
    print(e)
```

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:

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

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

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

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

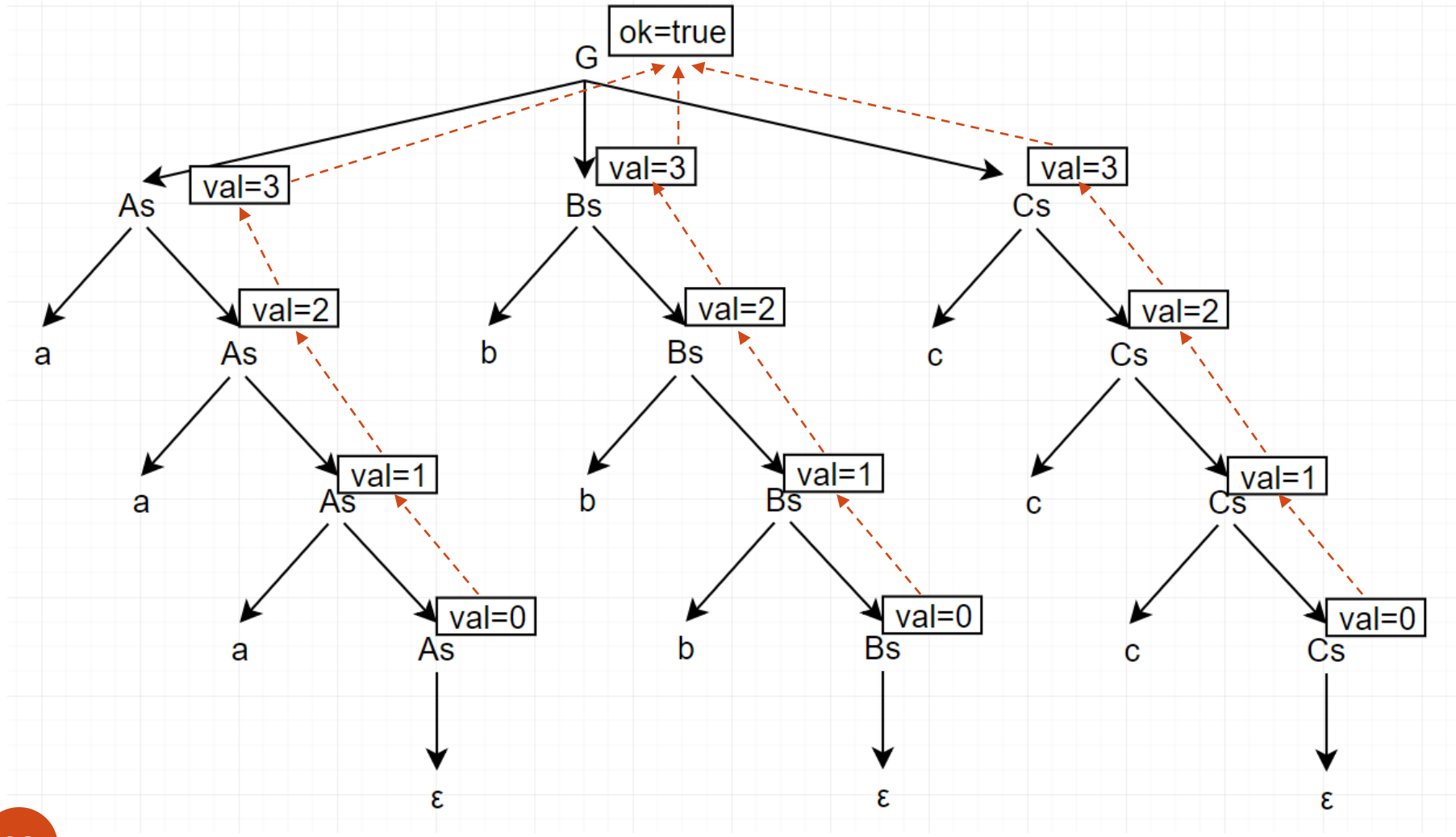
$$Bs \rightarrow \varepsilon \triangleright Bs.val := 0$$

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

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

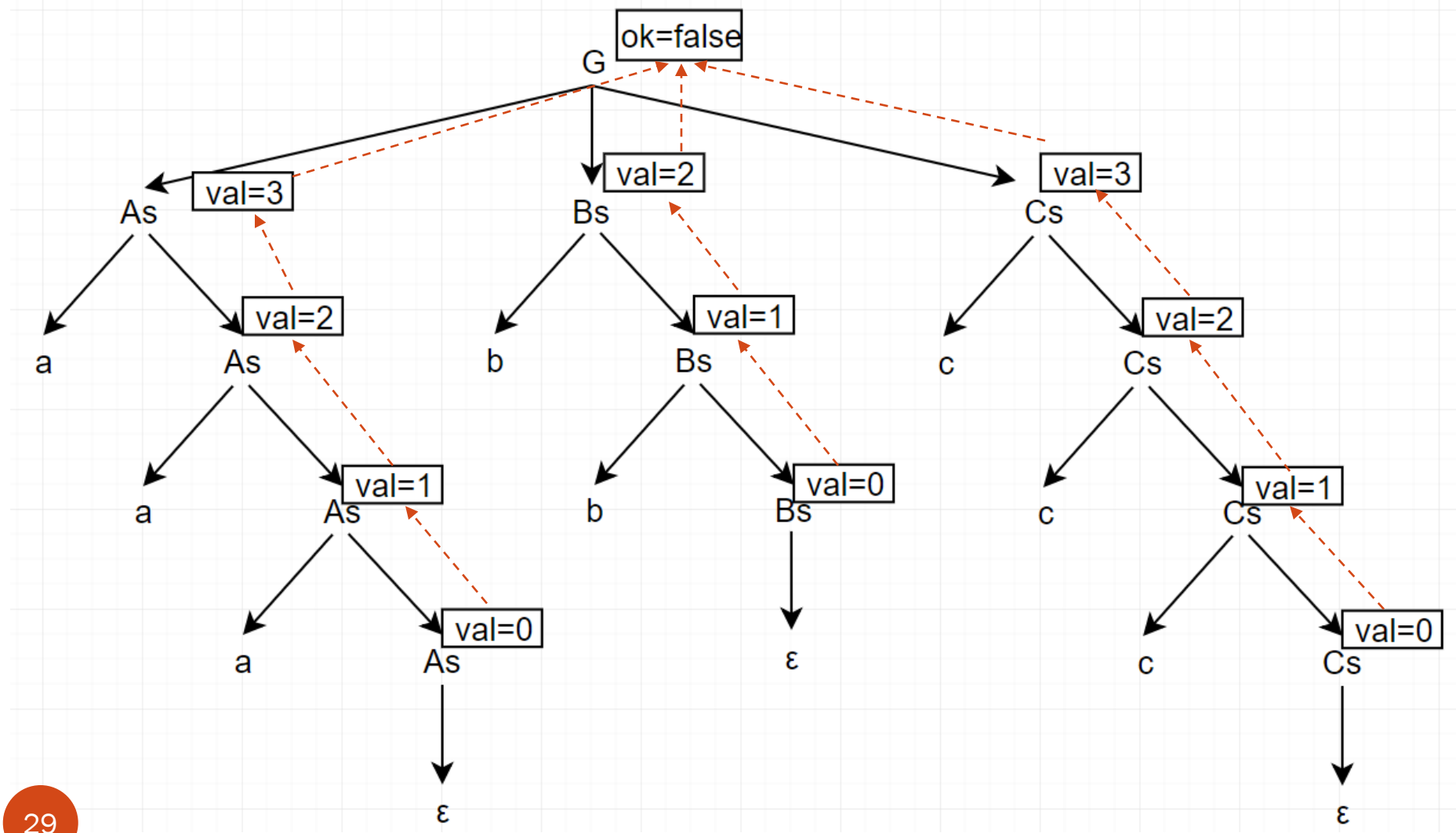
More than just CFG

- Annotate tree for "**aaabbbaaa**":



More than just CFG

- Annotate tree for "**aaabbaaa**":



Ply in python

```
# Tokens
tokens = (
    'A','B','C'
    )
t_A = r'a'
t_B = r'b'
t_C = r'c'
# Ignored characters
t_ignore = " \t"
def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
# Build the lexer
import ply.lex as lex
lex.lex()
```

Ply in python

```
# Parsing rules
def p_stmt(t):
    'statement : as bs cs'
    t[0] = t[1]==t[2] and t[2]==t[3]

def p_As(t):
    'as : A as'
    t[0] = 1 + t[2]
def p_As2(t):
    'as : '
    t[0] = 0

def p_Bs(t):
    'bs : B bs'
    t[0] = 1 + t[2]
def p_Bs2(t):
    'bs : '
    t[0] = 0
```

Ply in python

```
def p_Cs(t):
    'cs : C cs'
    t[0] = 1 + t[2]
def p_Cs2(t):
    'cs : '
    t[0] = 0
def p_error(t):
    print("Syntax error at '%s'" % t.value)

import ply.yacc as yacc
yacc.yacc()

try:
    s = "aaabbccc"
    a = yacc.parse(s)
    print(a)
except EOFError as e:
    print(e)
```

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

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

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
▷ E_1.val := sum(E_2.val, T.val)

$E_1 \longrightarrow E_2$ - T
▷ E_1.val := difference(E_2.val, T.val)

$E \longrightarrow T$
▷ E.val := T.val

copy rule

$T_1 \longrightarrow T_2$ * F
▷ T_1.val := product(T_2.val, F.val)

$T_1 \longrightarrow T_2$ / F
▷ T_1.val := quotient(T_2.val, F.val)

$T \longrightarrow F$
▷ T.val := F.val

$F_1 \longrightarrow$ - F_2
▷ F_1.val := additive_inverse(F_2.val)

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

$F \longrightarrow$ `const`
▷ F.val := const.val

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:

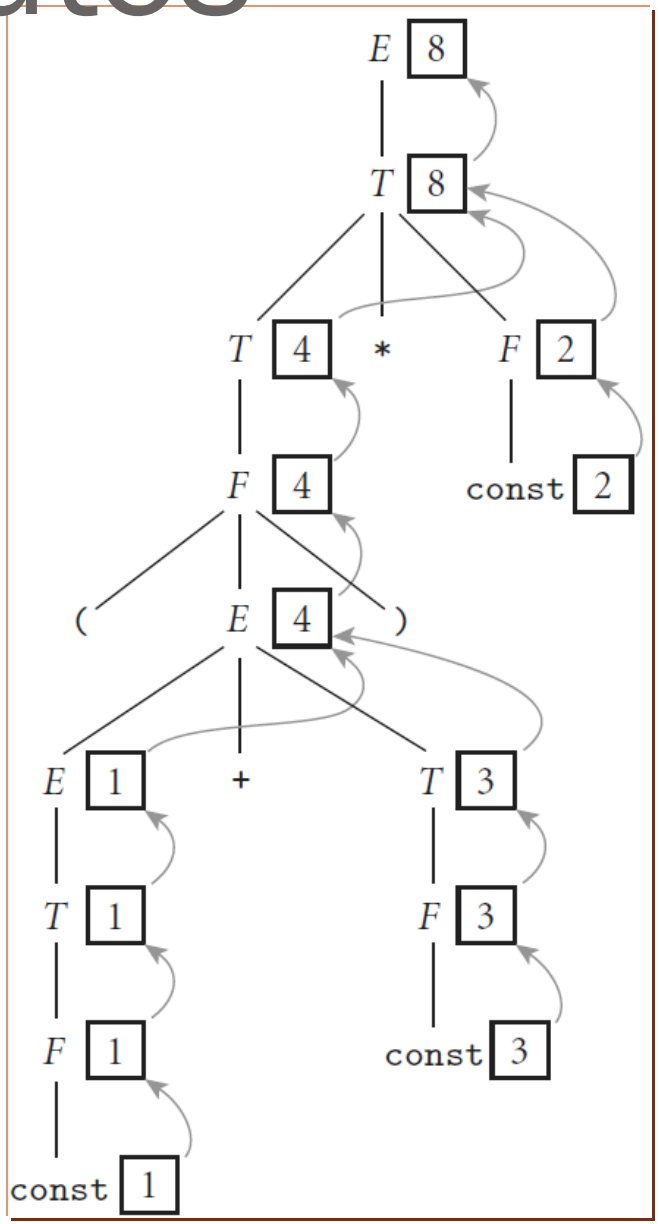
```
E1.val = sum(E2.val, T.val)
```

  - 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.



Ply in python

```
tokens = (
    'NAME','NUMBER',
    'PLUS','MINUS','TIMES','DIVIDE','EQUALS',
    'LPAREN','RPAREN',
    )
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_EQUALS  = r'='
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
t_NAME    = r'[a-zA-Z_][a-zA-Z0-9_]*'
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
t_ignore = " \t"
```

Ply in python

```
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
import ply.lex as lex
lex.lex()

# Parsing rules
precedence = (
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE')
    )

# dictionary of names
names = { }

def p_statement_assign(t):
    'statement : NAME EQUALS expression'
    names[t[1]] = t[3]
```

Ply in python

```
def p_statement_expr(t):
    'statement : expression'
    print(t[1])
def p_expression_binop(t):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''
    if t[2] == '+'  : t[0] = t[1] + t[3]
    elif t[2] == '-': t[0] = t[1] - t[3]
    elif t[2] == '*': t[0] = t[1] * t[3]
    elif t[2] == '/': t[0] = t[1] / t[3]
def p_expression_number(t):
    'expression : NUMBER'
    t[0] = t[1]
def p_expression_name(t):
    'expression : NAME'
    t[0] = names[t[1]]
```

Ply in python

```
def p_error(t):
    print("Syntax error at '%s'" % t.value)

import ply.yacc as yacc
yacc.yacc()

code = "1+2*3"
try:
    lex.input(code)
    while True:
        token = lex.token()
        if not token: break
        print(token)
    a = yacc.parse(code)
    print(a)
except Exception:
    print("ERROR")
```

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"

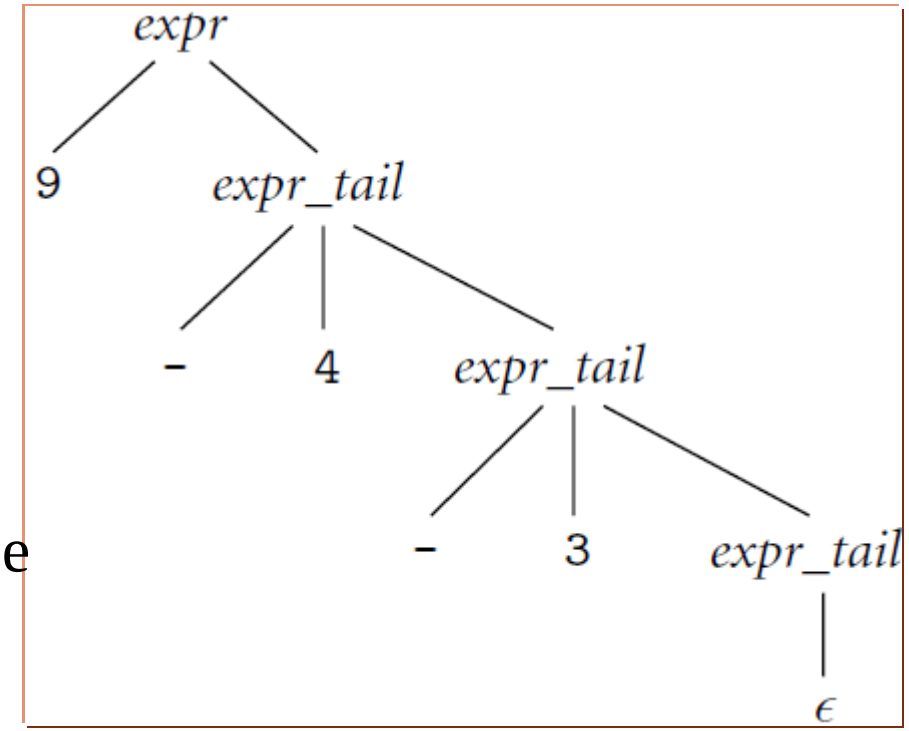
Evaluating Attributes

- 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



Synthesized Subtraction

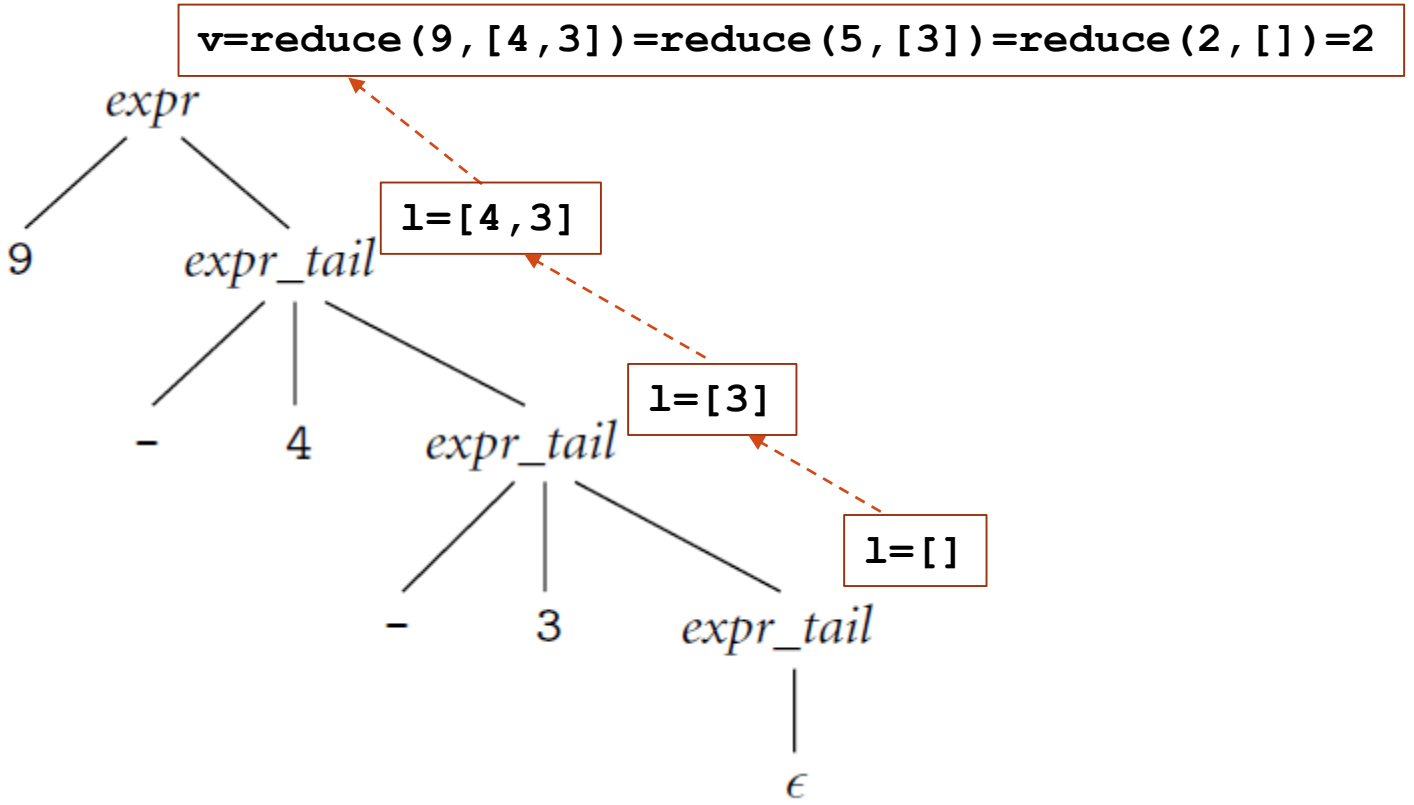
- We could also implement subtraction as synthesized, but we have to pass the **list** as the value of all subtracted terms and then use a semantic function to compute the result as the root attribute

```
expr -> const expr_tail
      ▷ expr.v := reduce(const.v, expr_tail.l)
expr_tail1 -> - const expr_tail2
      ▷ expr_tail1.l := const.v :: expr_tail2.l
expr_tail -> ε
      ▷ expr_tail.l := []
```

- SML semantic function:

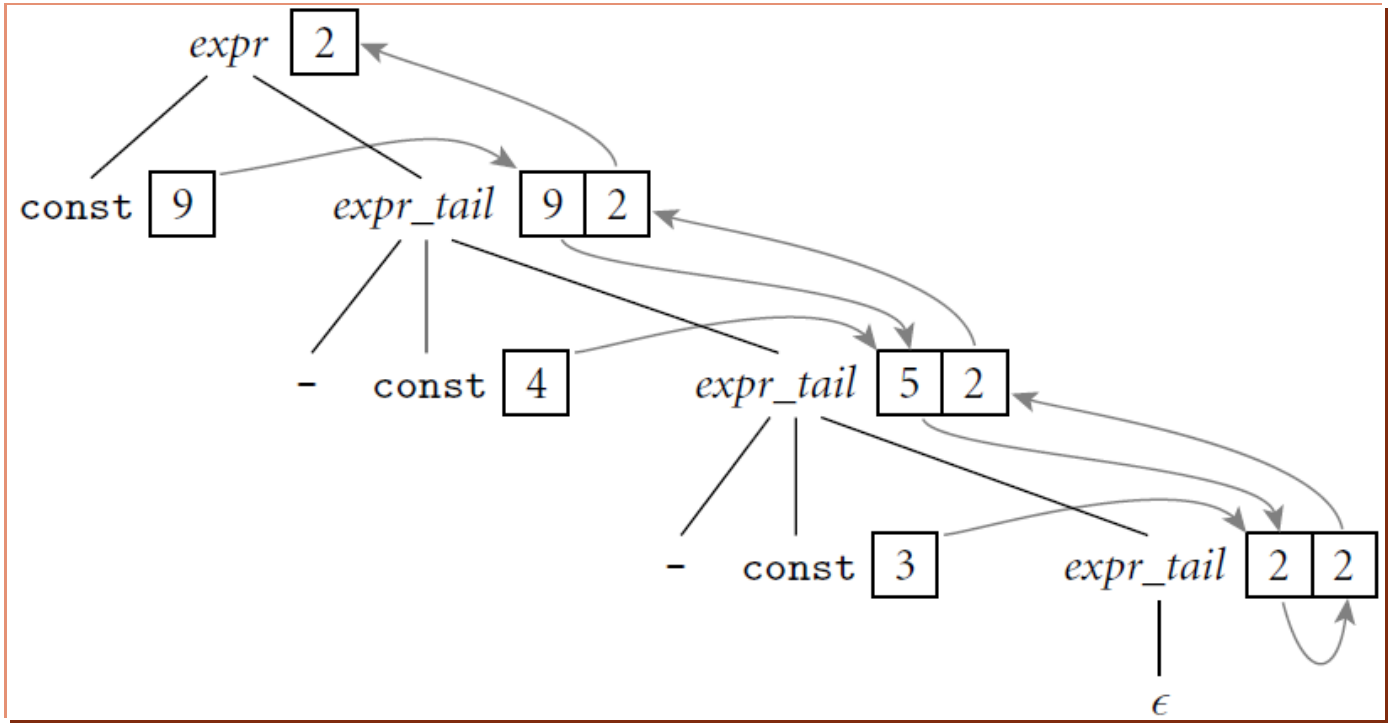
```
fun reduce(val, list) =
      if list = [] then val
      else reduce(val-hd(list),tl(list)))
```

Synthesized Subtraction



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 ⟶ `const` *expr_tail*
- ▷ expr_tail.st := const.val (1)
- ▷ expr.val := expr_tail.val (2)

$expr\_tail_1$ ⟶ `-` `const` $expr\_tail_2$
- ▷ $expr\_tail_2$.st := $expr\_tail_1$.st − const.val
- ▷ $expr\_tail_1$.val := $expr\_tail_2$.val

expr_tail ⟶ ϵ
- ▷ 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.

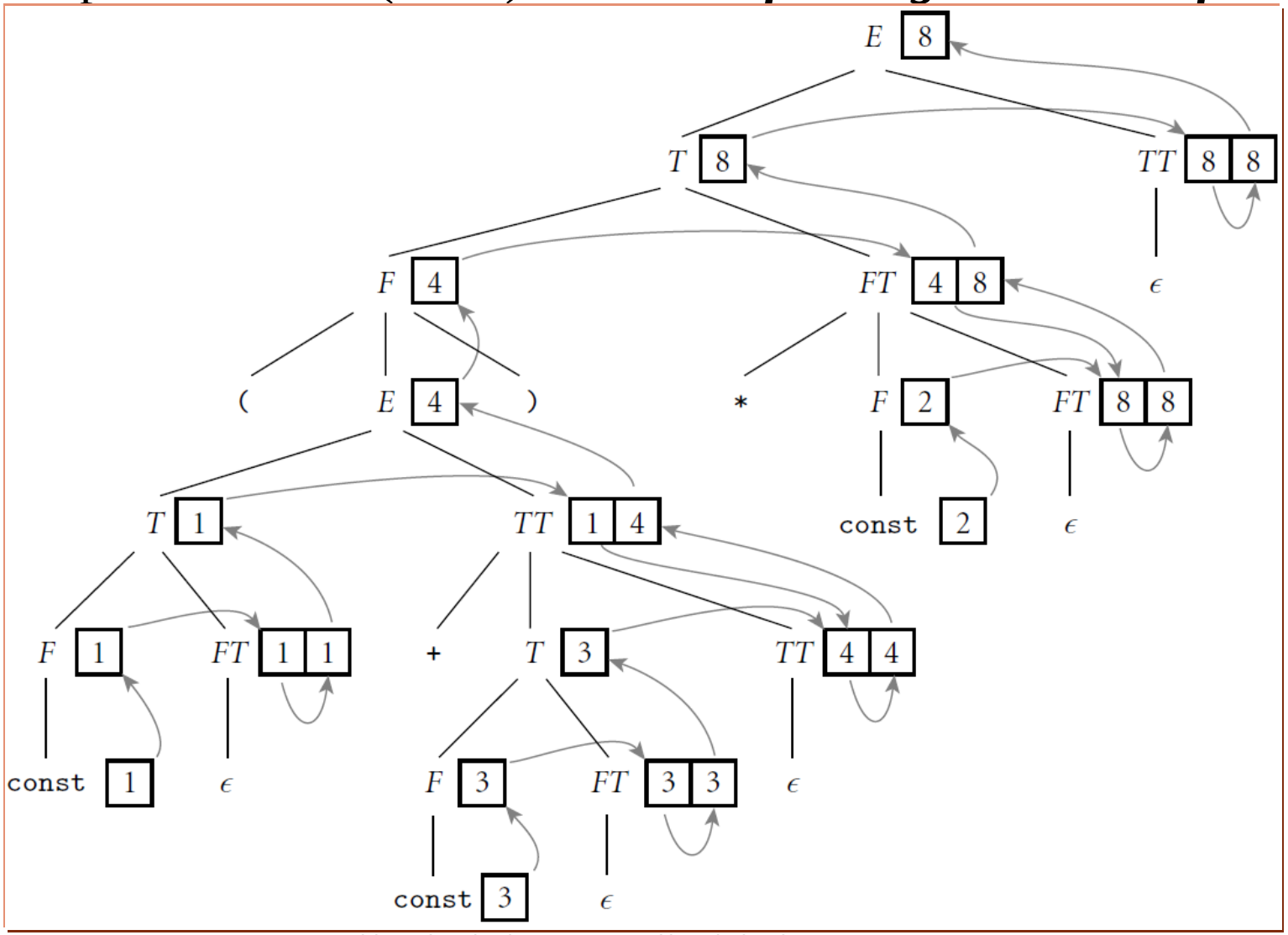
Evaluating Attributes

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

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

Evaluating Attributes

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



Evaluating Attributes

- 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
- 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

Evaluating Attributes

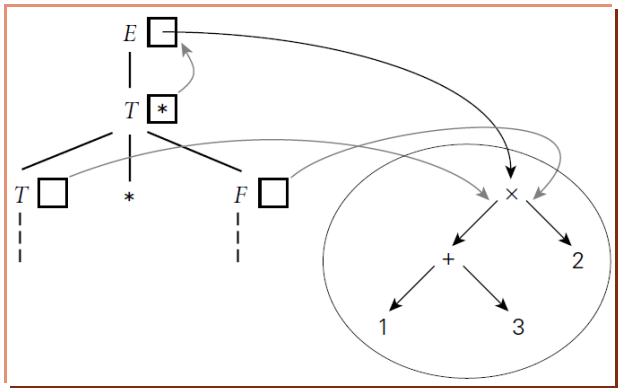
- ***Synthesized Attributes*** (in ***S-attributed grammars***):
  - Data flows bottom-up
  - Can be parsed by LR grammars
- ***Inherited Attributes***:
  - Data flows top-down and bottom-up
  - An attribute grammar is ***L-attributed*** if its attributes can be evaluated by visiting the nodes of the parse tree in a single left-to-right, depth-first traversal (same order with a top-down parse of an LL grammar)

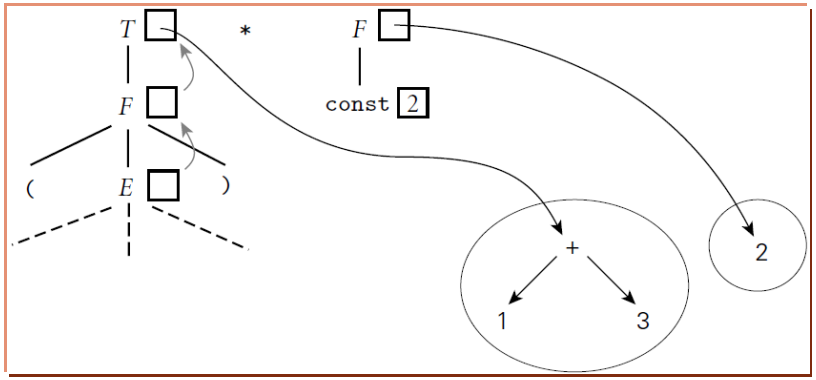
Syntax trees

- 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 a ***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
- If semantic analysis and code generation are **interleaved** with parsing, then action routines can be used to perform semantic checks and generate code directly
  - Later compilation phases can then consist of ad-hoc tree traversal(s)

Syntax trees

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





$E_1 \longrightarrow E_2 \; + \; T$
▷ E_1.ptr := make_bin_op("+", E_2.ptr, T.ptr)

$E_1 \longrightarrow E_2 \; - \; T$
▷ E_1.ptr := make_bin_op("–", E_2.ptr, T.ptr)

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

$T_1 \longrightarrow T_2 \; * \; F$
▷ T_1.ptr := make_bin_op("×", T_2.ptr, F.ptr)

$T_1 \longrightarrow T_2 \; / \; F$
▷ T_1.ptr := make_bin_op("÷", T_2.ptr, F.ptr)

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

$F_1 \longrightarrow - \; F_2$
▷ 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)

Ply in python

```
class Node:
    def __init__(self):
        print("init node")
    def evaluate(self):
        return 0
    def execute(self):
        return 0

class NumberNode(Node):
    def __init__(self, v):
        if('.' in v):
            self.value = float(v)
        else:
            self.value = int(v)
    def evaluate(self):
        return self.value
```

Ply in python

```
class BopNode(Node):
    def __init__(self, op, v1, v2):
        self.v1 = v1
        self.v2 = v2
        self.op = op
    def evaluate(self):
        if (self.op == '+'):
            return self.v1.evaluate() +
                self.v2.evaluate()
        elif (self.op == '-'):
            return self.v1.evaluate() -
                self.v2.evaluate()
        elif (self.op == '*'):
            return self.v1.evaluate() *
                self.v2.evaluate()
        elif (self.op == '/'):
            return self.v1.evaluate() /
                self.v2.evaluate()
```

Ply in python

```
class PrintNode(Node):
    def __init__(self, v):
        self.value = v
    def execute(self):
        print(self.value.evaluate())

tokens = (
    'PRINT',SEMI',
    'NUMBER',
    'PLUS','MINUS','TIMES','DIVIDE'
    )
# Tokens
t_PRINT   = 'print'
t_SEMI    = r';'
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
```

Ply in python

```
def t_NUMBER(t):
    r'-?\d*(\d\.|\.\d)\d* | \d+'
    try:
        t.value = NumberNode(t.value)
    except ValueError:
        print("Integer value too large %d", t.value)
        t.value = 0
    return t

# Ignored characters
t_ignore = " \t"

def t_error(t):
    print("Syntax error at '%s'" % t.value)

# Build the lexer
import ply.lex as lex
lex.lex()
```

Ply in python

```
# Parsing rules
precedence = (
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE')
    )

def p_print_smt(t):
    """
    print_smt : PRINT LPAREN expression RPAREN SEMI
    """
    t[0] = PrintNode(t[3])

def p_expression_binop(t):
    '''expression : expression PLUS factor
                  | expression MINUS factor
                  | expression TIMES factor
                  | expression DIVIDE factor'''
    t[0] = BopNode(t[2], t[1], t[3])
```

Ply in python

```
def p_expression_factor(t):
    '''expression : factor'''
    t[0] = t[1]

def p_factor_number(t):
    'factor : NUMBER'
    t[0] = t[1]

def p_error(t):
    print("Syntax error at '%s'" % t.value)

import ply.yacc as yacc
yacc.yacc()
```

Ply in python

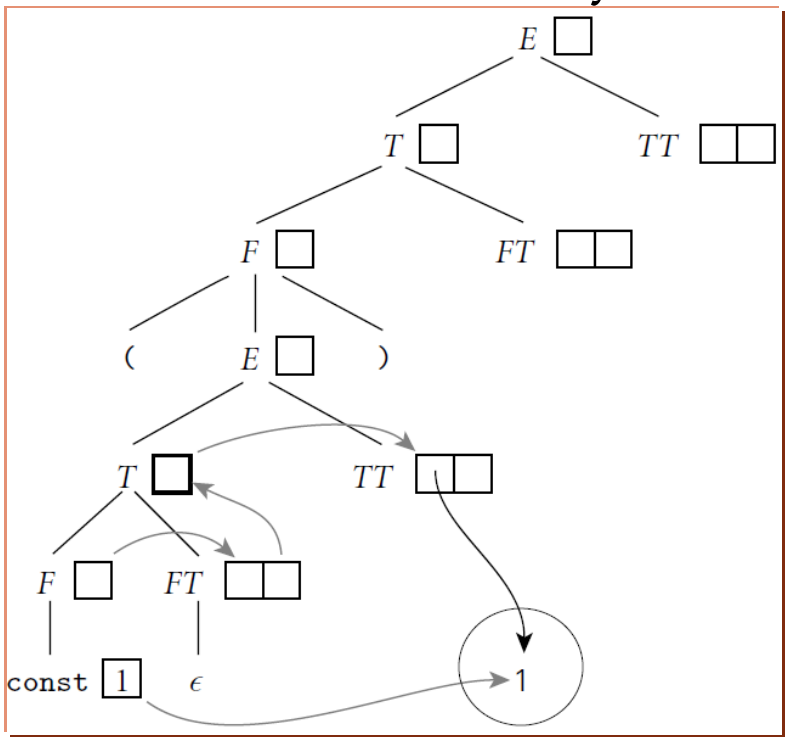
```
code = "print(1+3);"

try:
    lex.input(code)
    while True:
        token = lex.token()
        if not token: break
        print(token)
    ast = yacc.parse(code)
    print("AST: ",ast)
except Exception as e:
    print("Syntax Error: ",e)

try:
    ast.execute()
except Exception as e:
    print("Semantic Error: ",e)
```

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

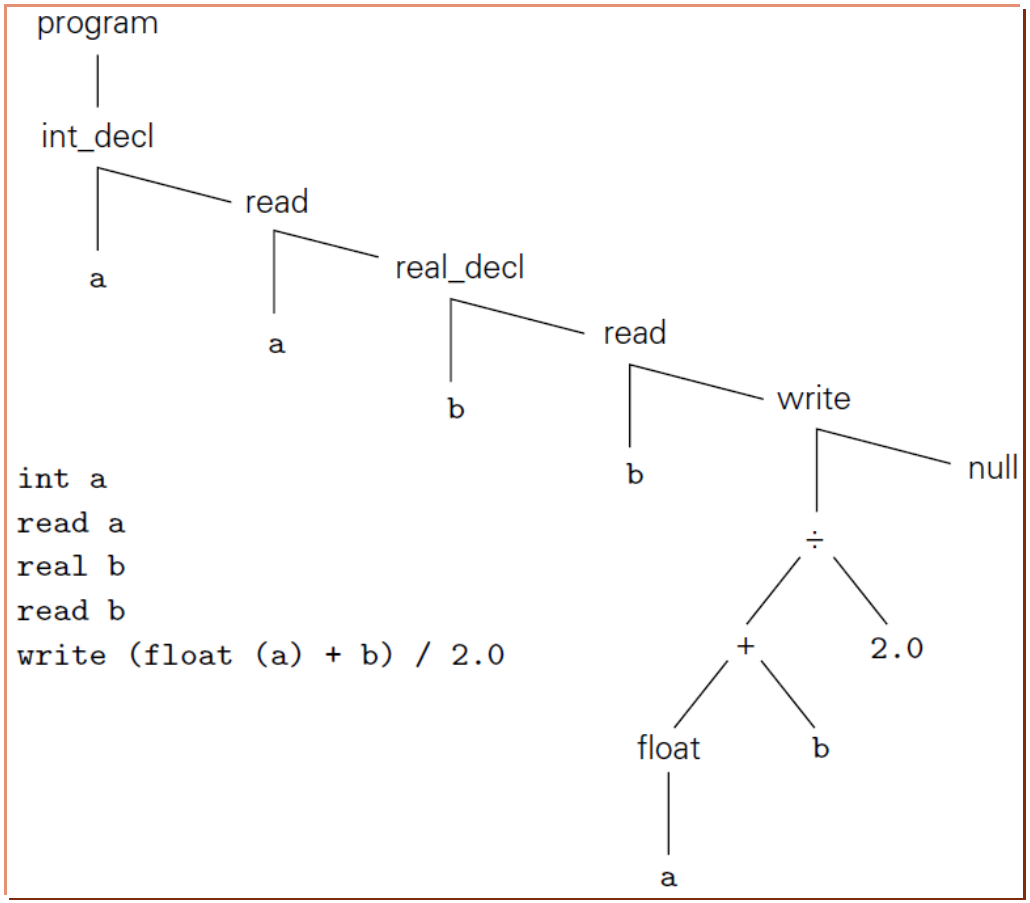
- While it is possible to construct automatic tools to analyze attribute flow and decorate parse trees, most compilers rely on ***action routines***, which the compiler writer embeds in the right-hand sides of productions to evaluate attribute rules at **specific points in a parse**
  - An ***action 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

Action Routines

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

program ⟶ *item*
int_decl : *item* ⟶ *id item*
read : *item* ⟶ *id item*
real_decl : *item* ⟶ *id item*
write : *item* ⟶ *expr item*

null : *item* ⟶ ϵ
'÷' : *expr* ⟶ *expr expr*
'+' : *expr* ⟶ *expr expr*
float : *expr* ⟶ *expr*
id : *expr* ⟶ ϵ
real_const : *expr* ⟶ ϵ



Decorating a Syntax Tree

- Sample of complete tree grammar representing structure of the syntax tree and use of a symbol table

id : *expr* ⟶ 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* ⟶ ϵ
  ▷ expr.type := int

real_const : *expr* ⟶ ϵ
  ▷ expr.type := real

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

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

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

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

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

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