# Semantic Analysis

Read: Scott, Chapter 4.1-4.3

# Announcements

- HW 1 grades are up
- Quiz 1,2,3 grades up
  - We will release answers next week

- Rainbow grades
  - Please check if your grade shows up correctly

trace.

gtrace.

# Lecture Outline

- Syntax vs. static semantics
- Static semantics vs. dynamic semantics

- Attribute Grammars
  - Attributes and rules
  - Synthesized and inherited attributes (next time)
  - S-attributed grammars (next time)
  - L-attributed grammars (next time)

# Static Semantics

- Earlier we considered syntax analysis
  - Informally, syntax deals with the form of programming language constructs
- We now look at static semantic analysis
  - Semantics deals with the meaning of programming language constructs
- The distinction between the two is fuzzy
  - In practice, anything that is not expressed in terms of certain CFG (LALR(1), in particular) is considered semantics

# Static Semantics

CFG:

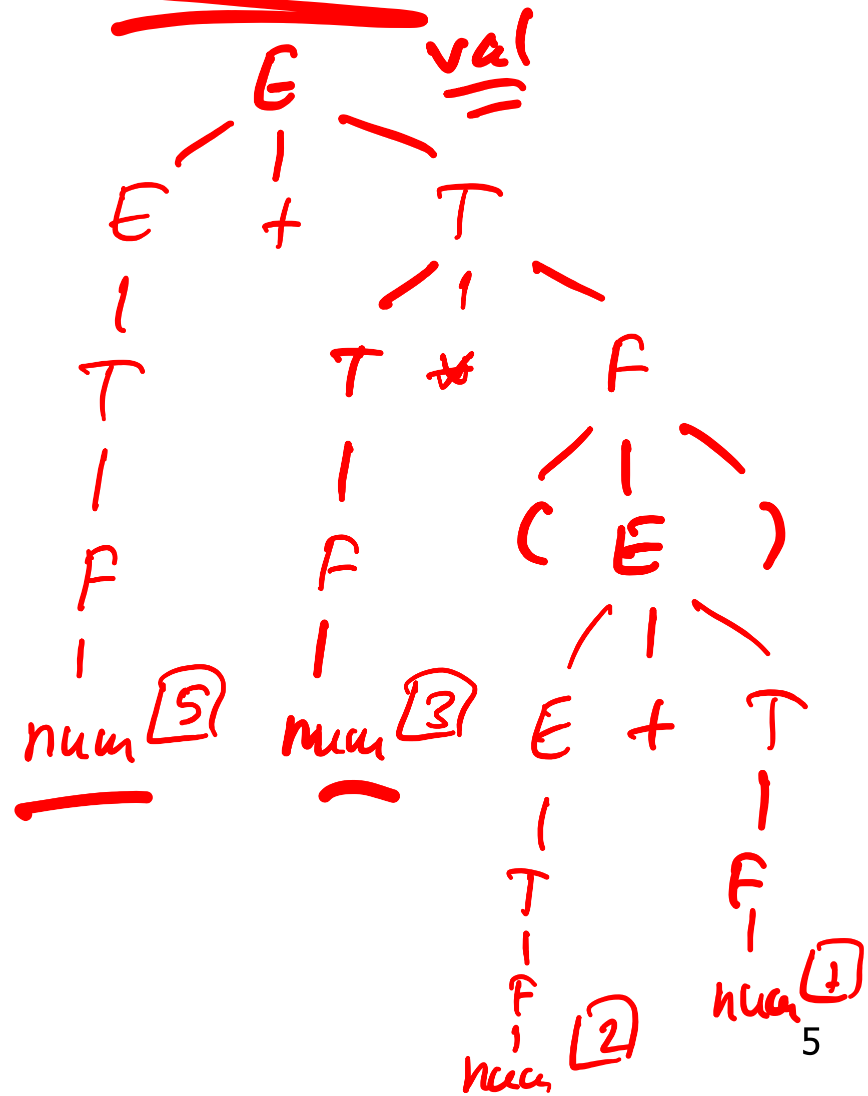
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow num \mid ( E )$

$5 + 3 * ( 2 + 1 )$

Parse tree:



# Static Semantics vs. Dynamic Semantics

- Static semantic analysis (compile-time)
  - Informally, reasons about program properties statically, before program execution
    - E.g., determine static types of expressions, detect certain errors
- Dynamic semantic analysis (run-time)
  - Reasons about program properties dynamically, during program execution
    - E.g., could expression `a[i]` index out of array bounds, etc.?

# The Role of Semantic Analysis

- Detect errors in programs!
- Static semantic analysis
  - Detect as many errors as possible early, before execution
    - Type inference and type checking
- Dynamic semantic analysis
  - Detect errors by performing checks during execution
    - Again, detect errors as early as possible. E.g., flagging an array-out-of-bounds at assignment `a[i] = …` is useful
    - Tradeoff: dynamic checks slow program execution
- Languages differ greatly in the amount of static semantic analysis and dynamic semantic analysis they perform

# Examples of Static Semantic Errors

- Type mismatch:
  - $type_x$ $type_e$
  - `x = y+z+w`: type of left-hand-side does not "match" type of right-hand-side
  - `A a; … ; a.m()`: `m()` cannot be invoked on a variable of type `A`
- Definite assignment check in Java: a local variable must be assigned before it is used

```
int x;
if (...)
  x=0;
y=x+1
```

# Examples of Dynamic Semantic Errors

- Null pointer dereference:
  - `a.m()` in Java, and `a` is null (i.e., uninitialized reference)
  - What happens?
- Array-index-out-of-bounds:
  - `a[i], i` goes beyond the bounds of `a`
  - What happens in C++? What happens in Java?
- Casting an object to a type of which it is not an instance
  - C++? Java?
- And more…

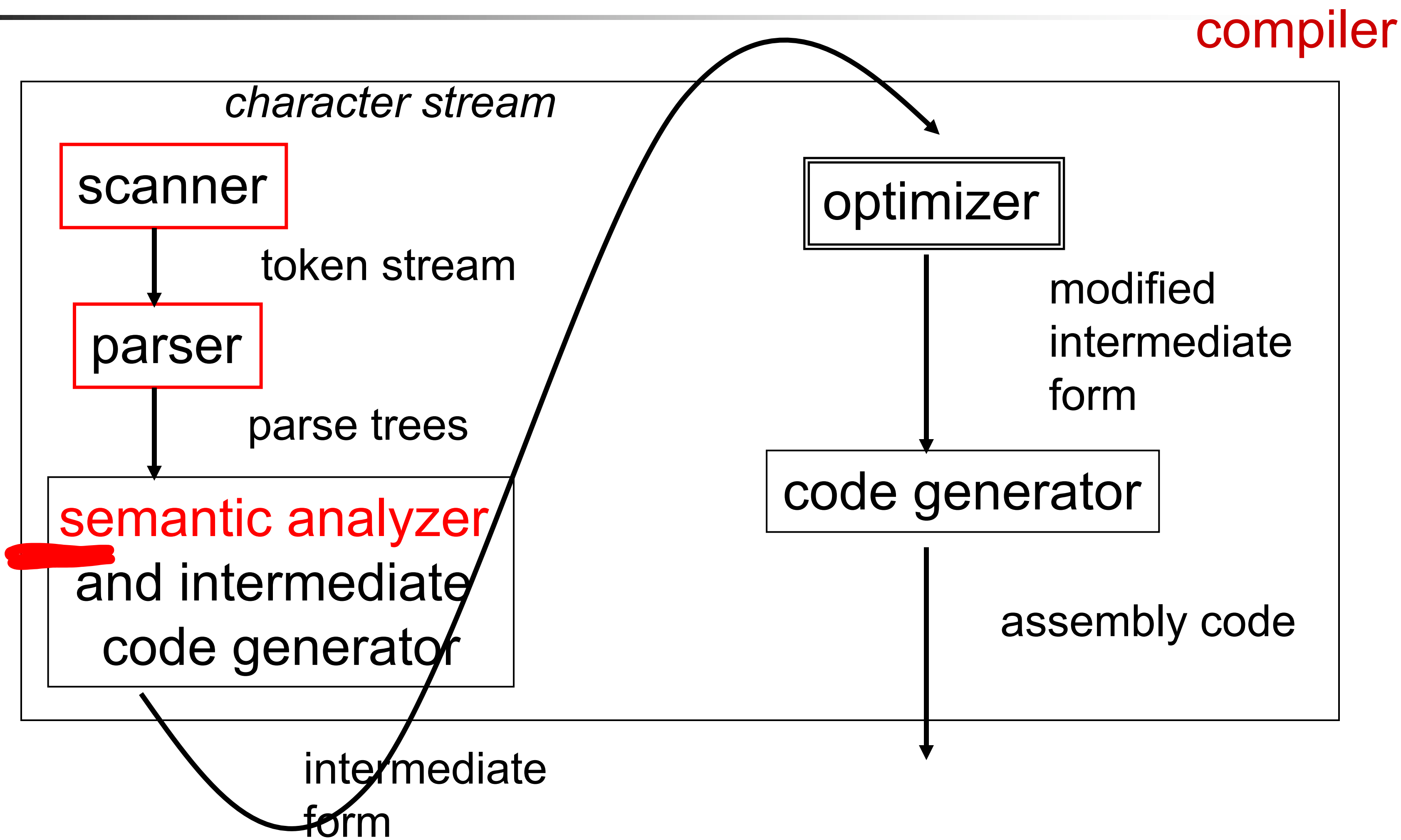
# Static Semantics vs. Dynamic Semantics

- Again, distinction between the two is fuzzy
- For some programs, the compiler **can** predict run-time behavior by using static analysis
  - E.g., there is no need for a nullness check:

```
x = new X();
x.m(); // x is non-null
```

- In general, the compiler cannot predict run-time behavior
  - Static analysis is limited by the halting problem

# Semantic Analyzer




Semantic analyzer performs static semantic analysis on parse trees and ASTs.
Optimizer performs static semantic analysis on intermediate 3-address code.

# Lecture Outline

- Syntax vs. static semantics
- Static semantics vs. dynamic semantics

Syntax-Directed Translation

- Attribute Grammars
  - Attributes and rules
  - Synthesized and inherited attributes (next time)
  - S-attributed grammars (next time)
  - L-attributed grammars (next time)

# Attribute Grammars: Foundation for Static Semantic Analysis

- Attribute Grammars: generalization of Context-Free Grammars
  - Associate meaning with parse trees
  - Attributes
    - Each grammar symbol has one or more values called attributes associated with it. Each parse tree node has its own instances of those attributes; attribute value carries the "meaning" of the parse tree rooted at node
  - Semantic rules
    - Each grammar production has associated rule, which may refer to and compute the values of attributes

# Example: Attribute Grammar to Compute Value of Expression (denote grammar by AG1)

- $S \to E \qquad E \to E + T \mid T \qquad T \to T * F \mid F \qquad F \to \mathbf{num}$

| Production | Semantic Rule |
|---|---|
| $S \to E$ | print($E.val$) |
| $E \to E_1 + T$ | $E.val := E_1.val + T.val$ |
| $E \to T$ | $E.val := T.val$ |
| $T \to T_1 * F$ | $T.val := T_1.val * F.val$ |
| $T \to F$ | $T.val := F.val$ |
| $F \to \mathbf{num}$ | $F.val := \mathbf{num}.val$ |

$E$ val =

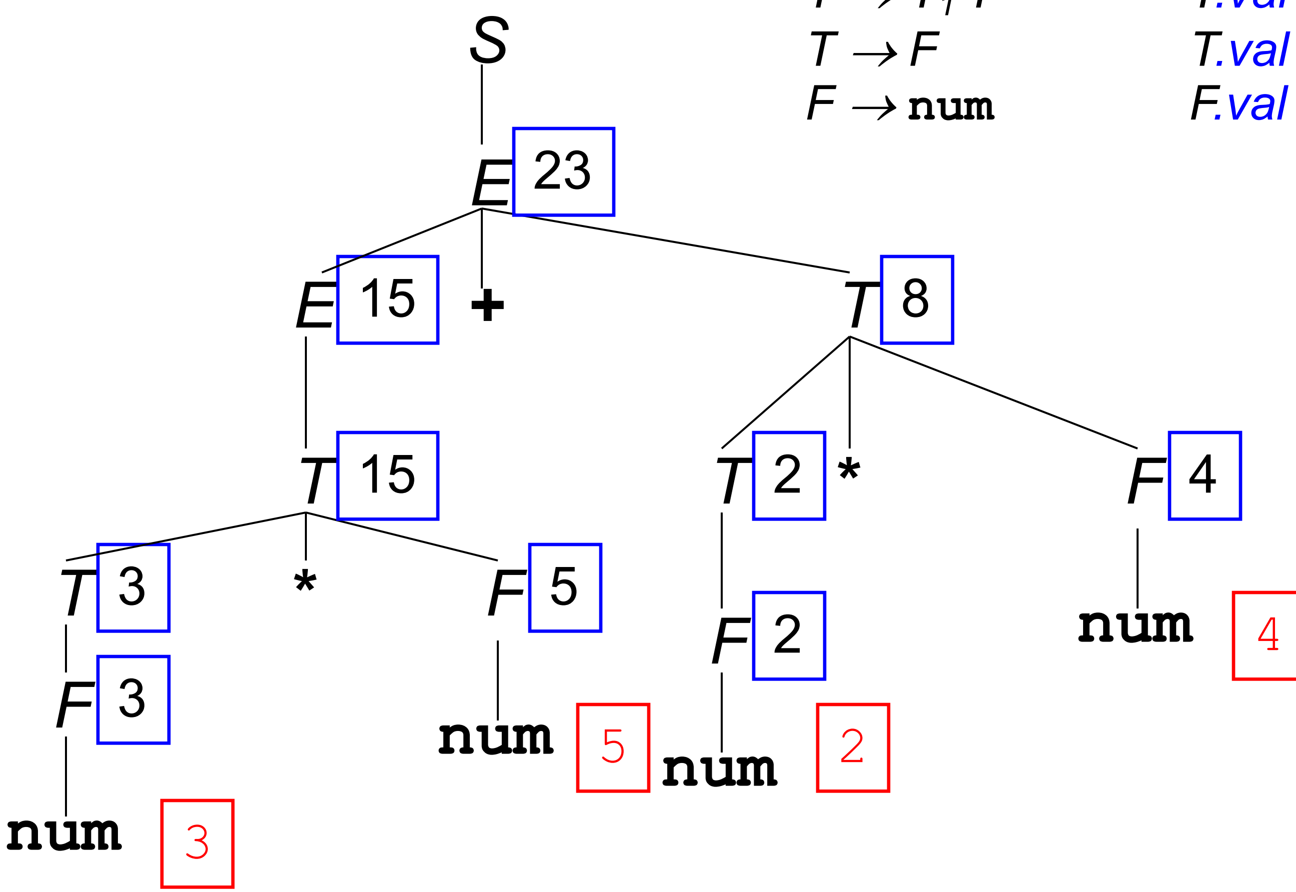
$E_1$ + $T$

..val

*val:* Attributes

# Example: Decorated parse tree for input `3*5 + 2*4`

| Production | Semantic rule |
|---|---|
| $S \to E$ | print($E.val$) |
| $E \to E_1 + T$ | $E.val := E_1.val + T.val$ |
| $E \to T$ | $E.val := T.val$ |
| $T \to T_1 * F$ | $T.val := T_1.val * F.val$ |
| $T \to F$ | $T.val := F.val$ |
| $F \to$ **num** | $F.val :=$ **num**$.val$ |



# Example

- *val:* Attributes associated to symbols
  - Intuitively, $A.val$ holds the value of the expression, represented by the subtree rooted at $A$
  - Separate attributes are associated with separate nodes in the parse tree
- Indices are used to distinguish between symbols with same name within same production
  - E.g., $E \rightarrow E_1+T$ $E.val := E_1.val+T.val$
- Attributes of terminals supplied by scanner
  - In example, attributes of **+** and ***** are never used

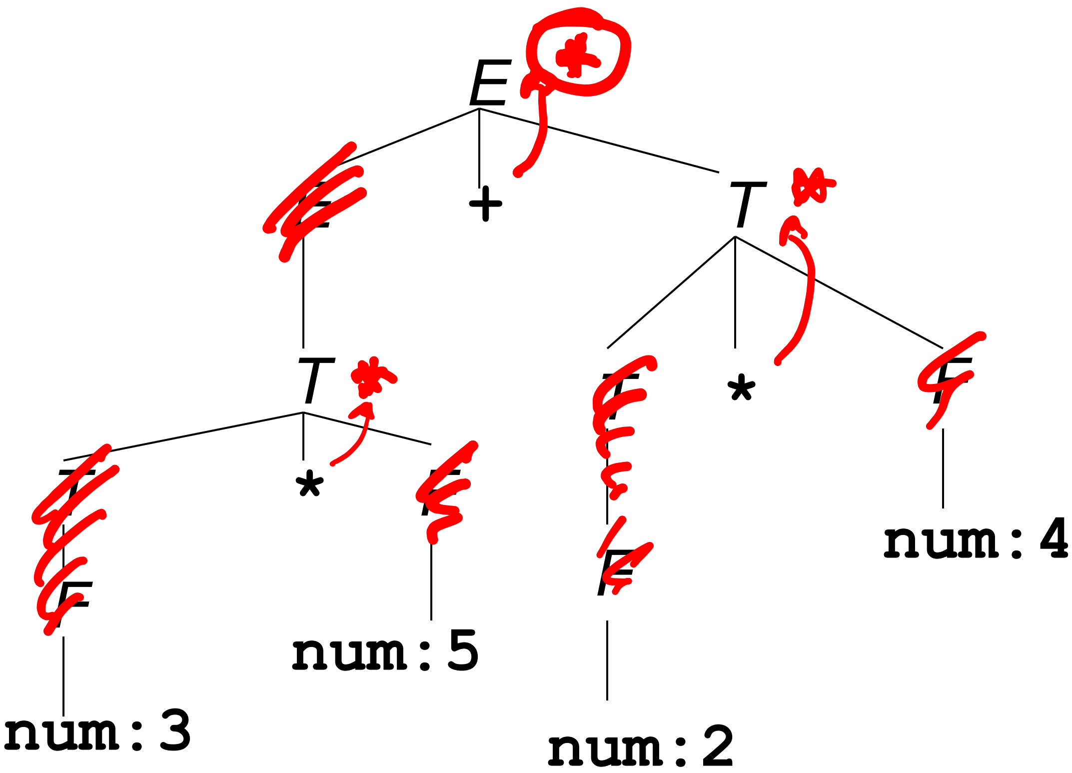
# Building an Abstract Syntax Tree (AST)

- An AST is an abbreviated parse tree
  - Operators and keywords do not appear as leaves, but at the interior node that would have been their parent
  - Chains of single productions are collapsed

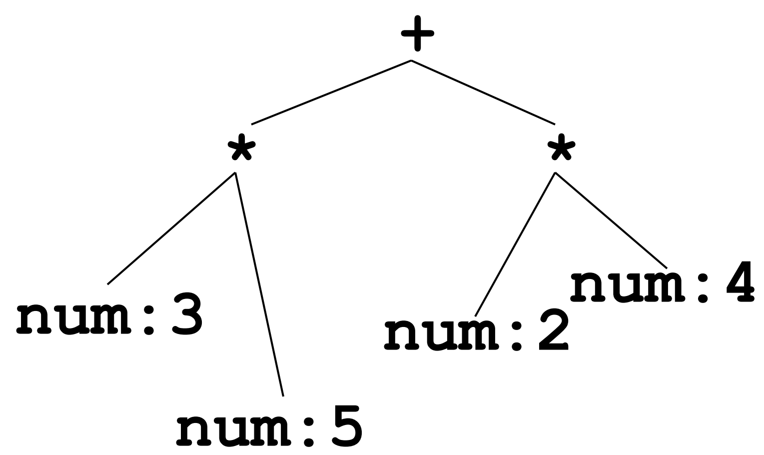
- Compilers typically work with ASTs

# Building ASTs for Expressions

Parse tree for **3*5+2*4**




Abstract syntax tree (AST)




How do we construct syntax trees for expressions?

# Attribute Grammar to build AST for Expression (denote by AG2)

- An attribute grammar:

Attribute “nodepointer” points to AST

| Production | Semantic Rule |
|---|---|
| $E \rightarrow E_1 + T$ | $E$.nptr := **mknode**(+, $E_1$.nptr, $T$.nptr) |
| $E \rightarrow T$ | $E$.nptr := $T$~~.nptr~~ |
| $T \rightarrow T_1 * F$ | $T$.nptr := **mknode**(*, $T_1$.nptr, $F$.nptr) |
| $T \rightarrow F$ | $T$.nptr := $F$.nptr |
| $F \rightarrow$ **num** | $F$.nptr := ***mkleaf***(**num**, **num**.*val*) |

***mknode(op,left,right)*** creates an operator node with label ***op***, and two fields containing pointers ***left,*** to left operand and ***right,*** to right operand

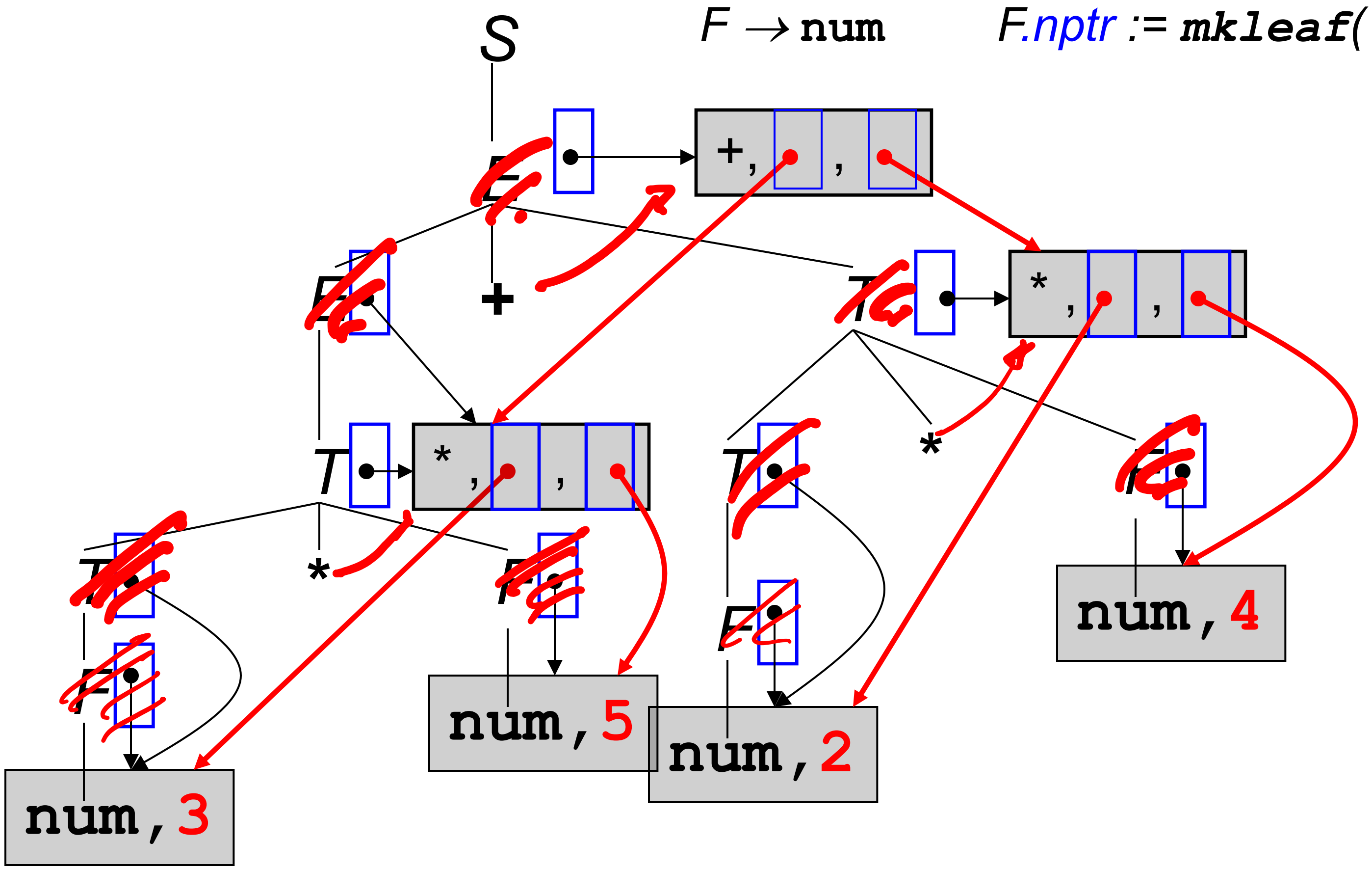
***mkleaf(num,num.****val****)*** creates a leaf node with label ***num***, and a field containing the value of the number

# Constructing ASTs for Expressions

Input:

**3 * 5 + 2 * 4**

| | |
|---|---|
| $E \rightarrow E_1 + T$ | $E.nptr :=$ **mknode**( '+', $E_1.nptr$, $T.nptr$) |
| $E \rightarrow T$ | $E.nptr := T.nptr$ |
| $T \rightarrow T_1 * F$ | $T.nptr :=$ **mknode**( '*', $T_1.nptr$, $F.nptr$) |
| $T \rightarrow F$ | $T.nptr := F.nptr$ |
| $F \rightarrow$ **num** | $F.nptr :=$ **mkleaf**('**num**', **num**.$val$) |



# Exercise

- We know that the language $L$ = $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ is not context free. It can be captured however with an attribute grammar. Give an underlying CFG and a set of attribute rules that associate an attribute **`ok`** with the root *S* of each parse tree, such that *S*.**`ok`** is true if and only if the string corresponding to the fringe of the tree is in *L*.

# Exercise

$a^n b^n c^n$

CFG:

$a^m b^n c^p$

Attribute Grammar

$S \to ABC$ (count, count, count)

if $A.count == B.count == C.count$: $S.ok = true$
else: $S.ok = false$

$A \to A_1 a$ — $A.count = A_1.count + 1$

$A \to \varepsilon$ — $A.count = 0$

$B \to B_1 b$ — $B.count = B_1.count + 1$

$B \to \varepsilon$ — $B.count = 0$

$C \to C_1 c$ — $C.count = C_1.count + 1$

$C \to \varepsilon$ — $C.count = 0$

# Exercise

- Consider the expression grammar

$$E \rightarrow E \textbf{ + } T \mid T$$
$$T \rightarrow T \textbf{ * } F \mid F$$
$$F \rightarrow \texttt{num} \mid \texttt{(} E \texttt{)}$$

Give attribute rules to accumulate into the root a count of the maximum depth to which parentheses are nested in the expression. E.g., ((1 + 2)*3 + 4)*5 + 6 has a count of 2.

# Exercise