Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 9: Grammars and SDT

Context-free grammars,
disambiguation,
syntax-directed translation

Announcements

Project proposal due this Wed

HW3 out today

- due this Sunday

PA3 out in two days

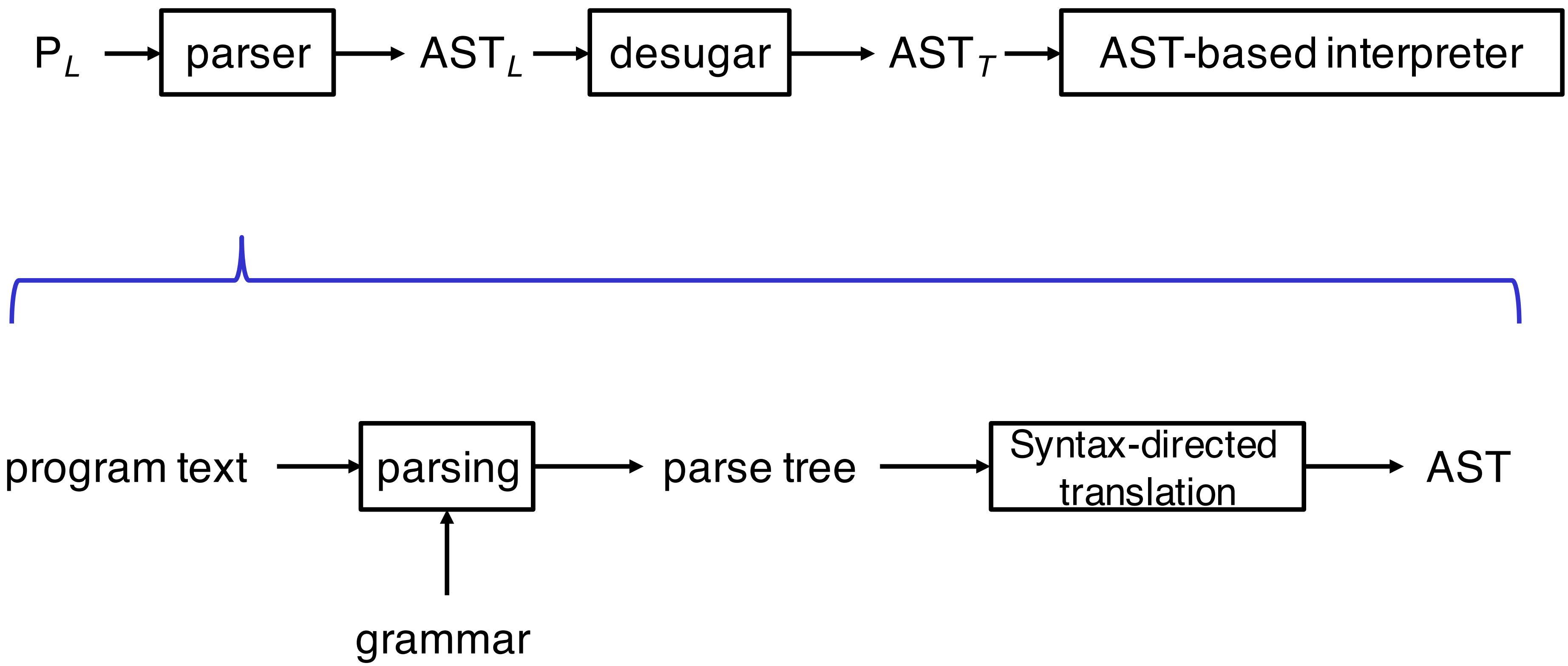
- be due in roughly 3 weeks

Announcements

Midterm exam in class on 2/10

- Covers everything including this week and HW3
- Closed book
- One letter-size sheet of **handwritten** notes (one sided)
- Previous exams on website
- Review session: Sunday 6-7pm EEB 115

Next two lectures will expand on the parser



Today

Parse tree

- The result of parsing a string with a grammar

Ambiguities

- handling grammars with ambiguous rules

Computations on parse trees

- Compilation, Interpretation, Type Checking, Doc Layout

Attribute grammar

- grammar where parse tree nodes have attributes

Syntax-directed evaluation

- Evaluation of the attributes of a parse tree

[Multi-pass attribute grammars (for layout)]

Grammars and parse trees

HW3.1a

A grammar has four components

Example grammar:

$$
\begin{aligned}
E \rightarrow\ & n \\
|\ & E + E \\
|\ & E * E \\
|\ & (E)
\end{aligned}
$$

- Nonterminals:
- Start nonterminal:
- Terminals:
- Productions (rules):

A grammar has four components

Example grammar:

$$
\begin{aligned}
E \rightarrow\ & n \\
\mid\ & E + E \\
\mid\ & E * E \\
\mid\ & (E)
\end{aligned}
$$

- Nonterminals: E
- Start nonterminal: E
- Terminals: n, +, *, (,)
- Productions (rules): $E \rightarrow n$, $E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow (E)$

Parse tree

Produced given a grammar and an input string

sometimes the parse tree is not built explicitly

represents the structure present in flat input strings

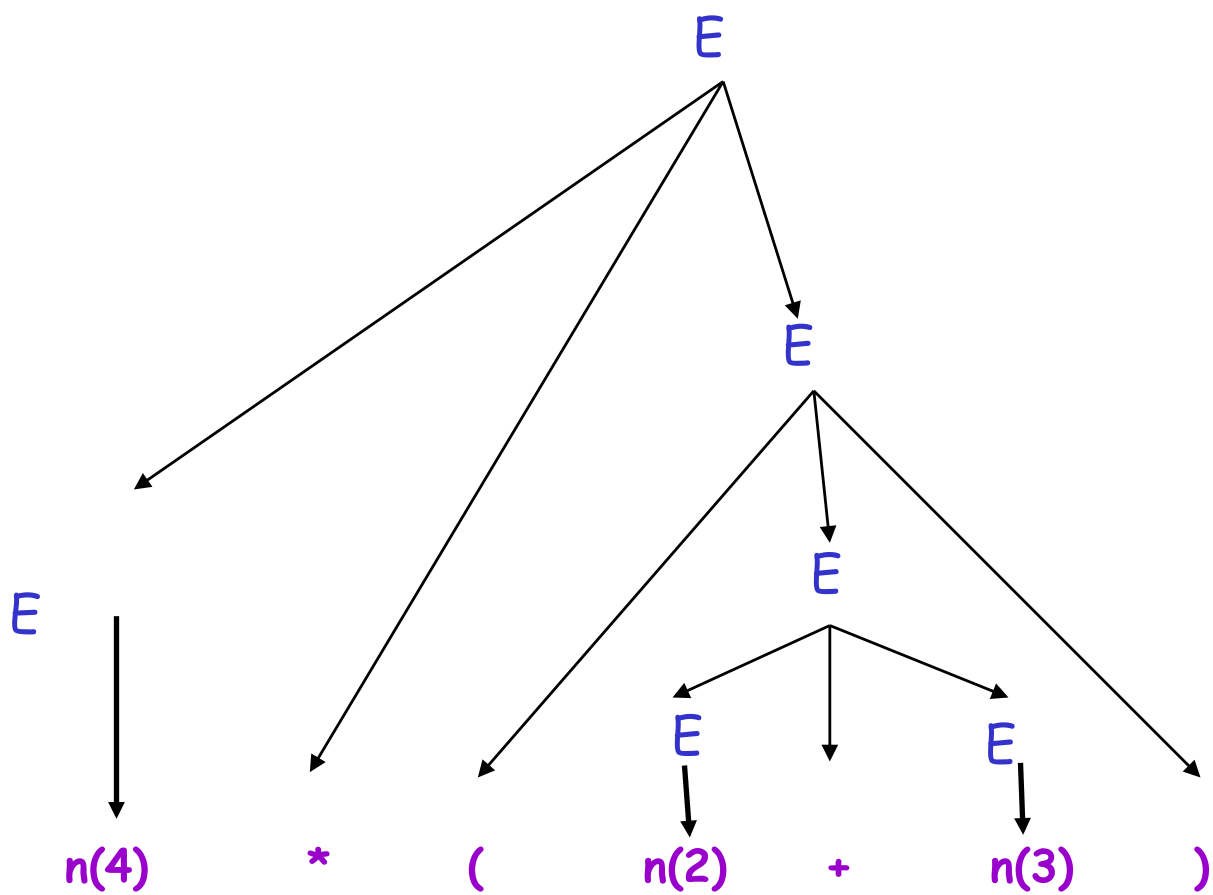
composed from productions used to derive the string

Parse tree example

Parser input: **n(4), *, (, n(2), +, n(3),)**

Parse tree:

$$
\begin{aligned}
E \rightarrow\ & n \\
|\ & E + E \\
|\ & E * E \\
|\ & (E)
\end{aligned}
$$



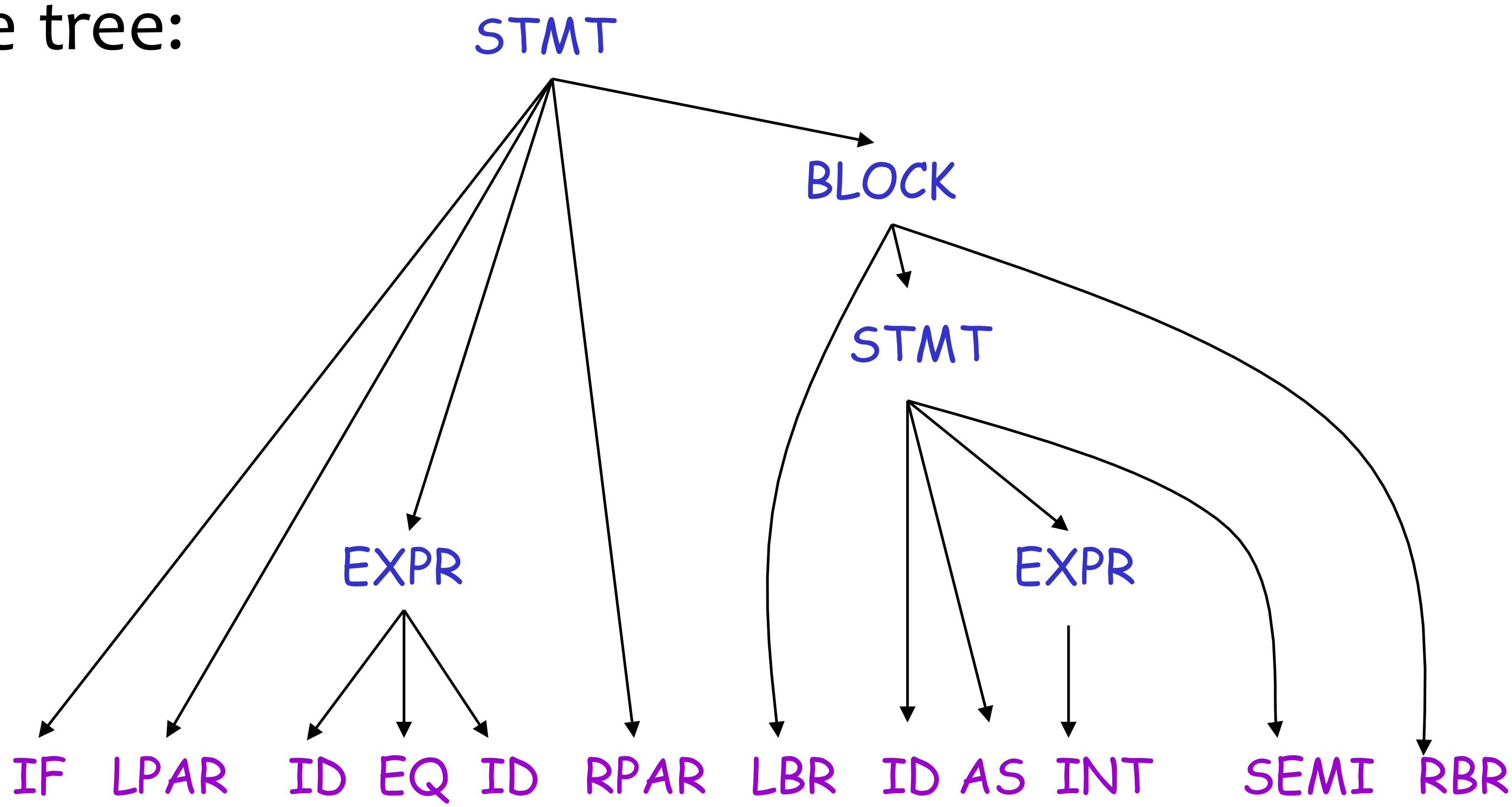
☞ leaves are tokens (terminals), internal nodes are non-terminals

Another example of parse tree

Source string: if (x == y) { a=1; }

Tokenized: **IF, LPAR, ID, EQ, ID, RPAR, LBR, ID, AS, INT, SEMI, RBR**

Parse tree:



Ambiguous Grammars

HW3.1b

Ambiguity

The shape of parse tree depends on the grammar

The grammar is designed so that the parse tree reflects desired operator precedence and associativity

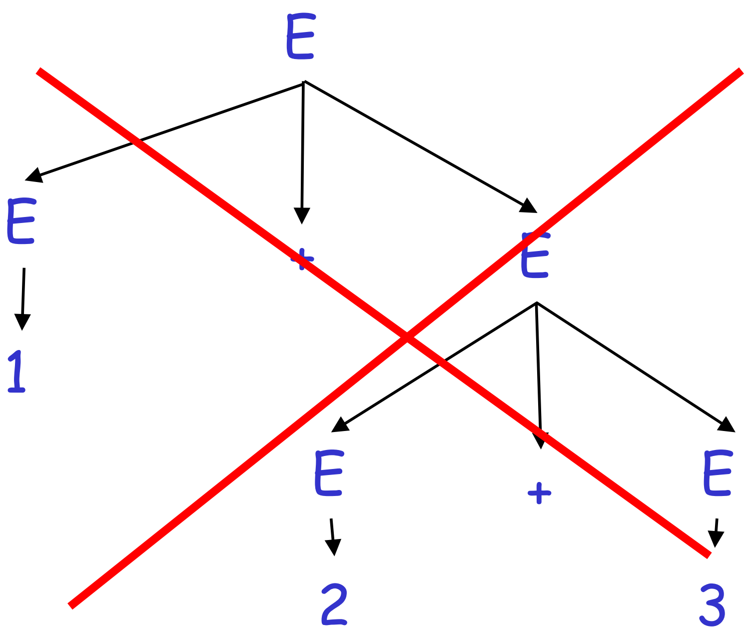
The arithmetic grammar we've seen permits multiple parse trees, so this grammar does NOT capture precedence, associativity

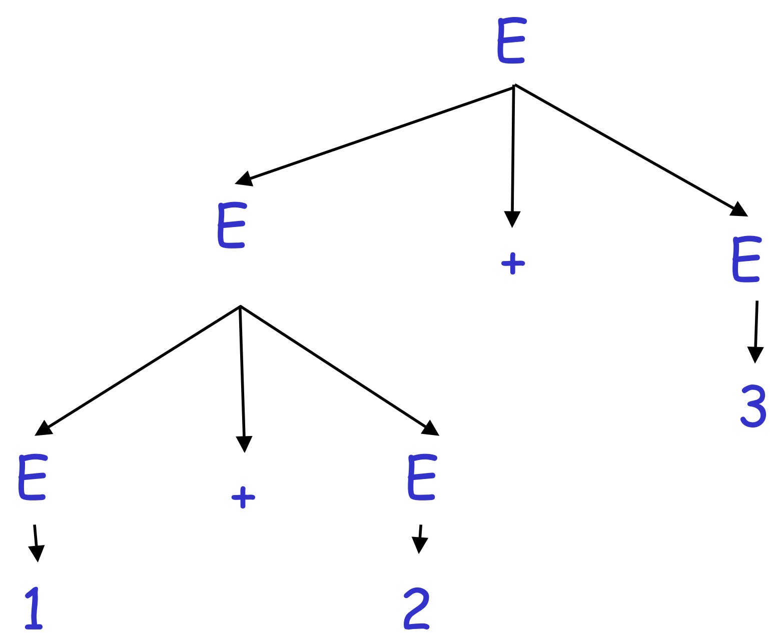
Example of multiple parse trees on next slide

Example of ambiguous grammar

E → E + E | *n*

Given input: 1 + 2 + 3:





+ is left associative

What is an ambiguous grammar?

Ambiguous grammar:

When a (any) string produces more than one parse tree

Why is this bad?

The meaning of the input is not defined

Disambiguation via Grammar Rewriting

HW3.1c

Rewriting

Rewrite the grammar into a unambiguous grammar

new grammar defines the same language (set of legal strings) but eliminates undesirable parse trees

Example: Rewrite

$$E \rightarrow E + E \mid E * E \mid (E) \mid n$$

into

$$E \rightarrow E + T \mid T$$

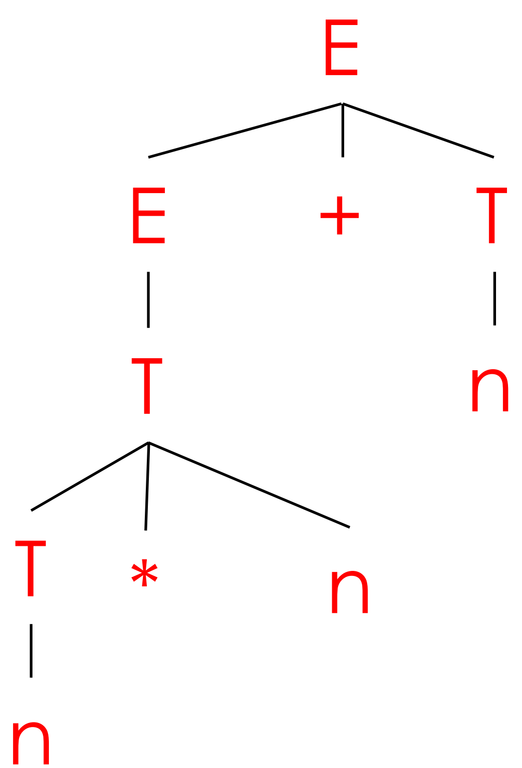
$$T \rightarrow T * n \mid n \mid (E)$$

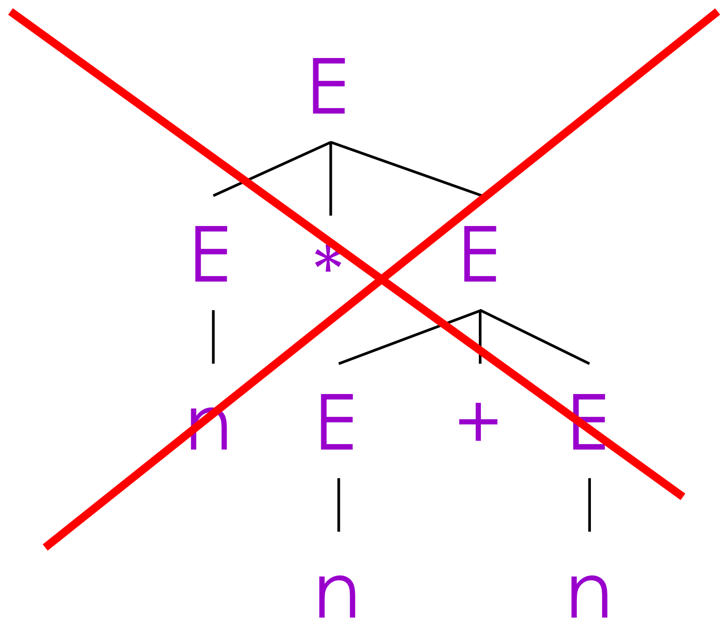
Draw a few parse trees and you will see that new grammar

- enforces precedence of * over +
- enforces left-associativity of + and *

Parse tree with the new grammar

The int * int + int has ony one parse tree now

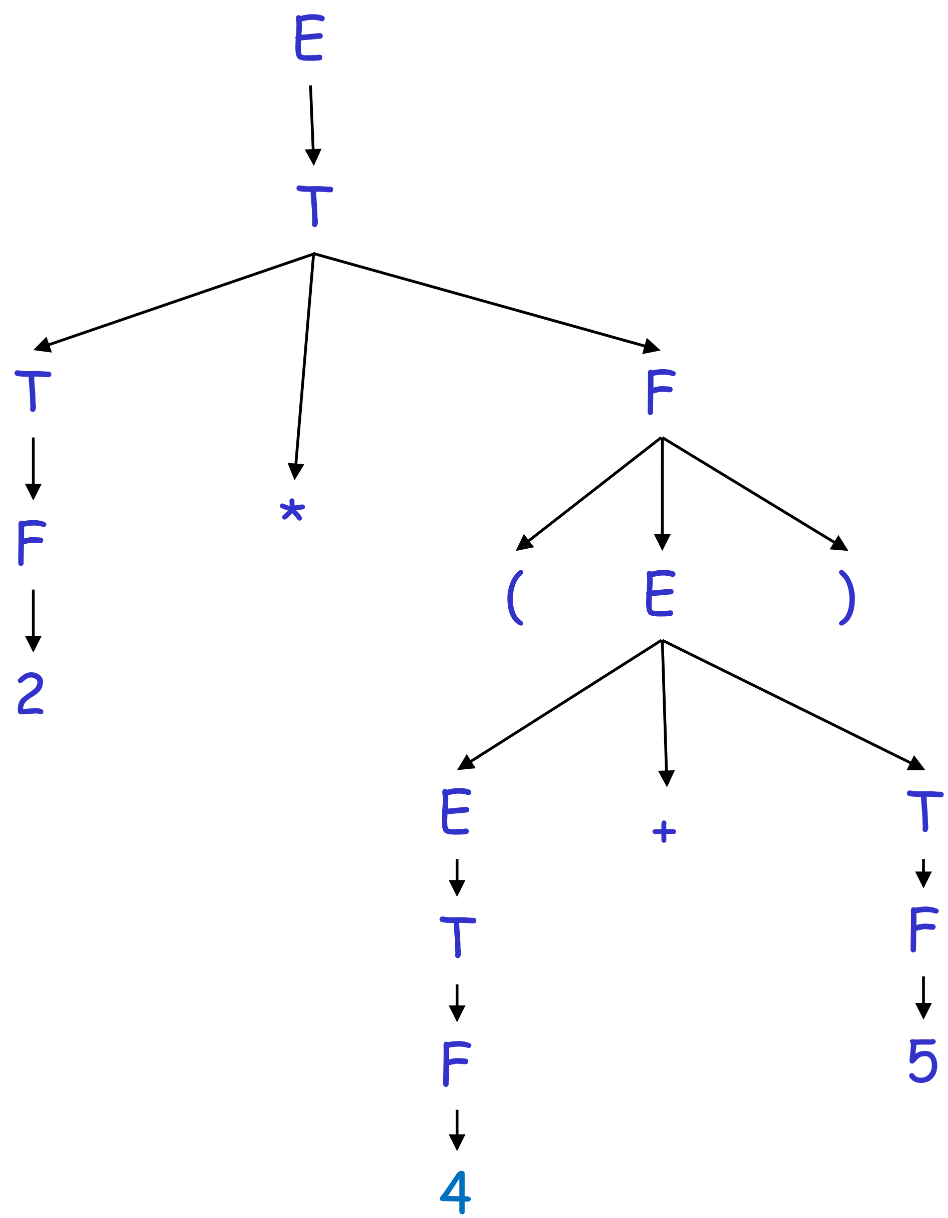




note that new nonterminals have been introduced

Usually, this grammar is written with E, T, F

E → E + T | T
T → T * F | F
F → (E) | n



Exercise

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid n$

Convince yourself that you cannot parse 2+3*4 to give + higher precedence than *

Some terminology

Language: set of strings

- L(G) – strings generated by grammar G
- L(N) – strings generated by non-terminal N

Left-recursive grammar: includes rule X → X …

- generates left-associative expressions

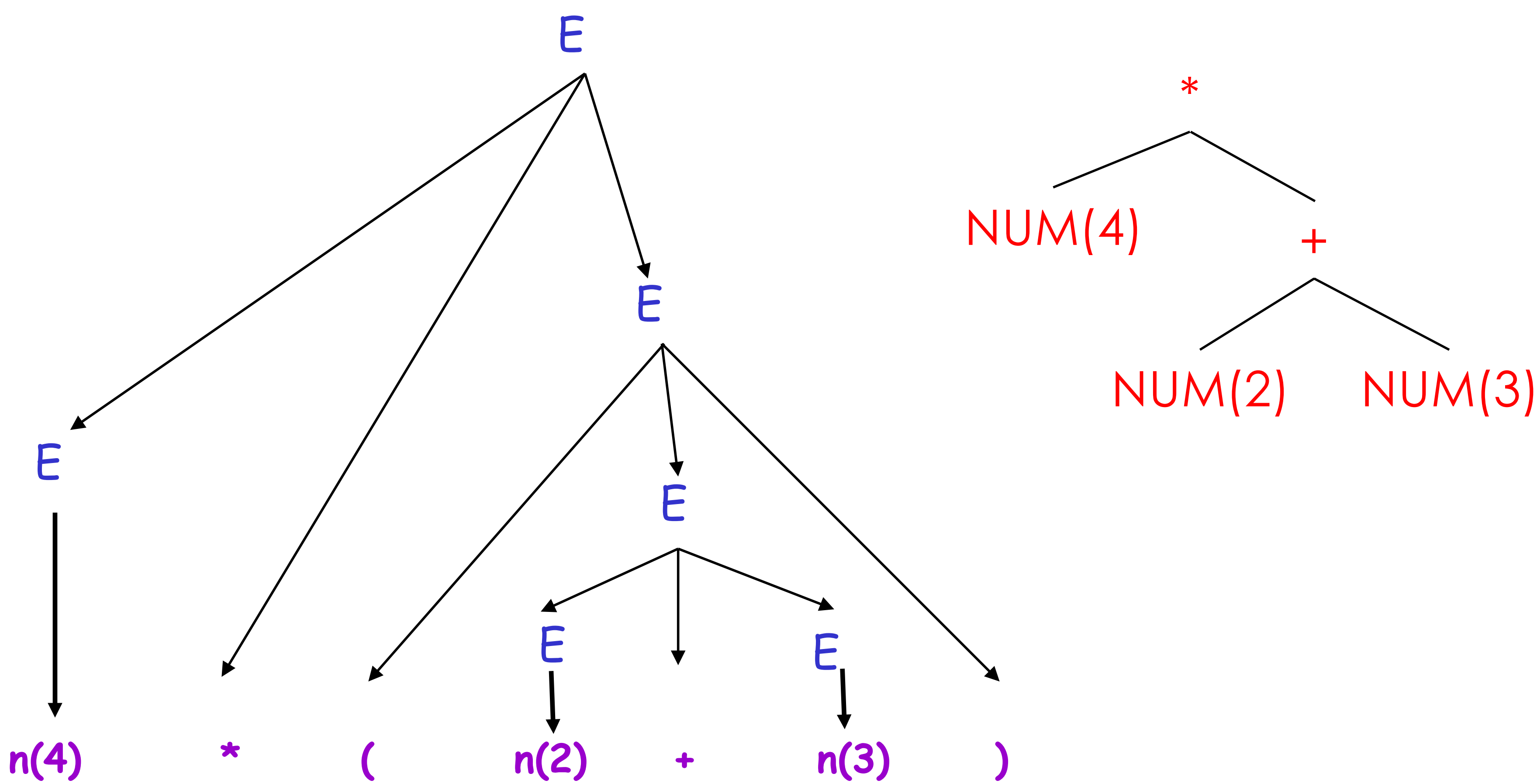
Right-recursive grammar: includes rule X → … X

- generates right-associative expressions

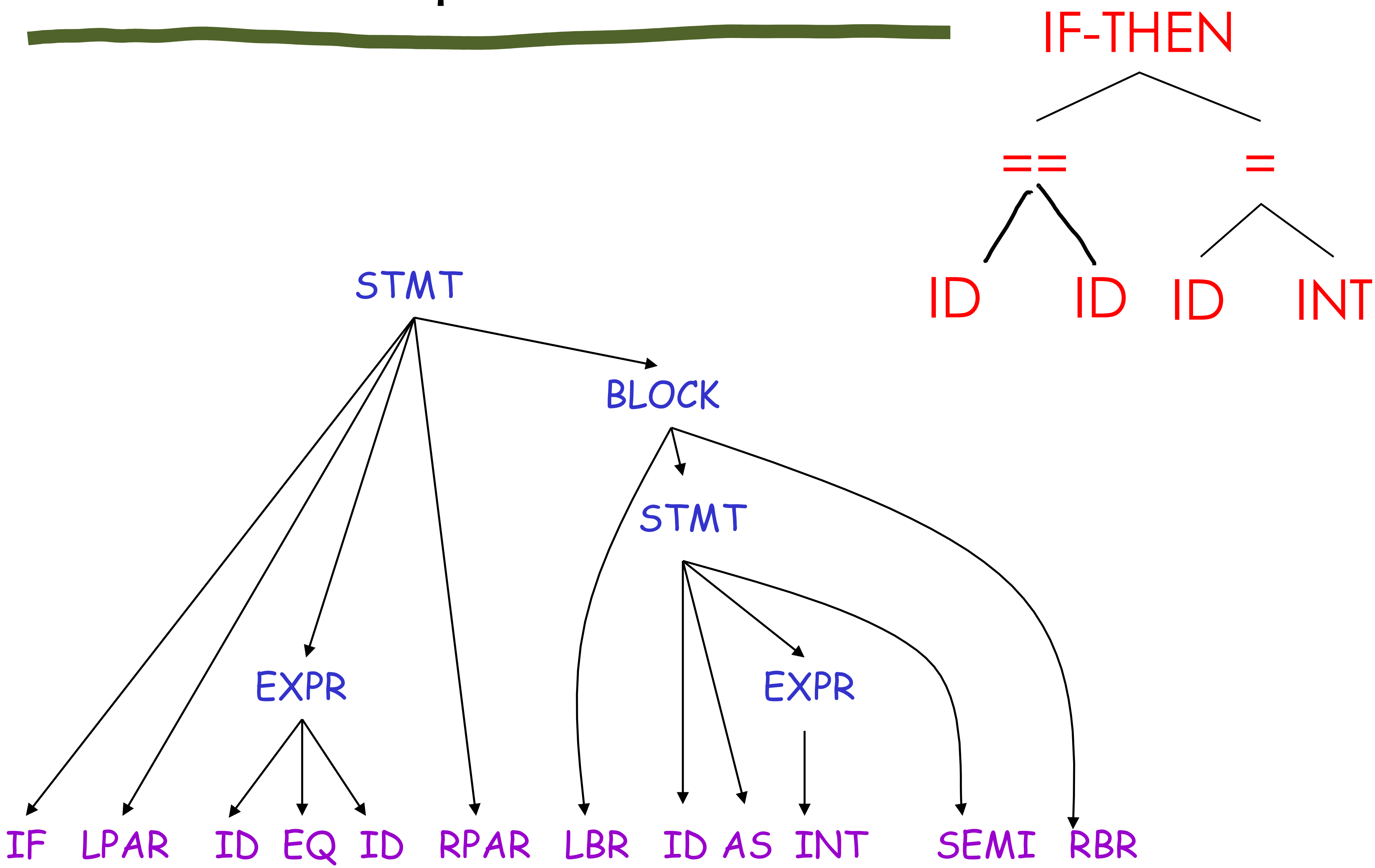
The Abstract Syntax Tree

a compact representation of the parse tree

AST is a compression of the parse tree



Another example



How to construct an AST?

By evaluating the parse tree! Key steps:

1. Extend nodes with attributes

   Such as val attribute for expression nodes

2. Specify how attributes are computed from other attrs

   These are assignments of the form
   `E1.val = E2.val + E3.val`

3. Determine the evaluation order

   How will these computations happen?

   Bottom up, top down, or inorder?

   (Most often, it's bottom-up.)

Grammars with attributes are *attribute grammars*

Attribute Grammars

What are attribute grammars?

- Grammars with attributes (duh!)

- Grammars with attributes stored at each node
  - With attribute values computed from other nodes

Two types of attributes

Synthesized: computed from children

Inherited: computed from parent (and siblings)

Attribute grammars

Applications of attribute grammars

- evaluate the input program P (interpret P)
- type check the program (look for errors before eval)
- construct AST of P (abstract the parse tree)

- generate code (which when executed, will evaluate P)
- compile (regular expressions to automata)

- document layout (compute positions, sizes of letters)
- programming tools (syntax highlighting)

An example attribute grammar (AG)

Idea: evaluate expressions by storing their values as attributes

Each node now comes with a "`val`" attribute

We now need to define rules for computing this attribute

E_1 ::= E_2 + T	E_1.val = E_2.val + T.val
E ::= T	E.val = T.val
T_1 ::= T_2 * F	T_1.val = T_2.val * F.val
T ::= F	T.val = F.val
F ::= n	F.val = n.val
F ::= (E)	F.val = E.val

Is `val` an inherited or synthesized attribute?

An example attribute grammar (AG)

$E \rightarrow E + T$	$E_1 ::= E_2 + T$	$E_1.val = E_2.val + T.val$
$\mid T$	$E ::= T$	$E.val = T.val$
$T \rightarrow T * F$	$T_1 ::= T_2 * F$	$T_1.val = T_2.val * F.val$
$\mid F$	$T ::= F$	$T.val = F.val$
$F \rightarrow n$	$F ::= n$	$F.val = n.val$
$\mid (E)$	$F ::= (E)$	$F.val = E.val$

AG = grammar + "semantic rules"
rules show how to evaluate parse tree

Same AG in 401 parser notation

AG for evaluating an expression

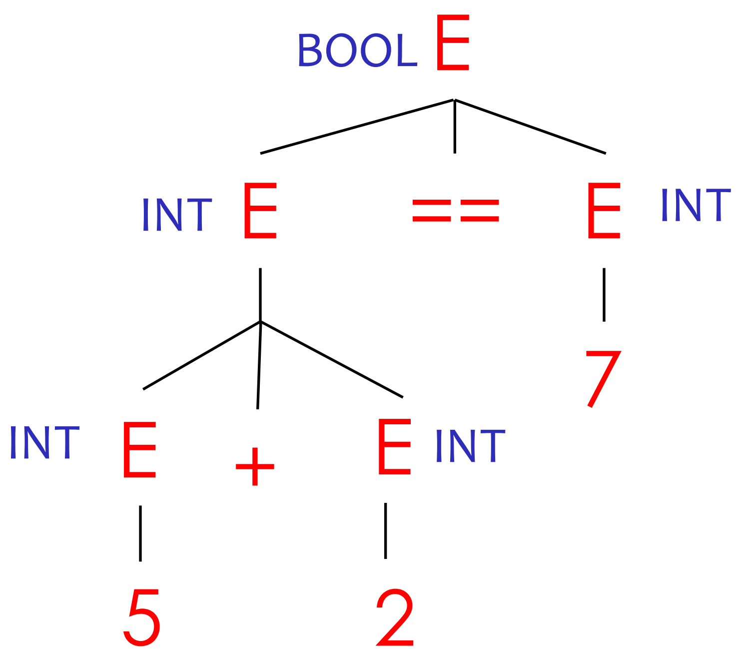
```
%%
E -> E '+' T    %{ return n1.val + n3.val }%
   |  T         %{ return n1.val }%
   ;
T -> T '*' F    %{ return n1.val * n3.val }%
   |  F         %{ return n1.val }%
   ;
F -> /[0-9]+/   %{ return int(n1.val) }%
   | '(' E ')'  %{ return n2.val }%
   ;
```

Compare this with our interpreter

Another AG: Compute type of expression + typecheck

```
E -> E + E    if ((E2.val == INT) and (E3.val == INT))
                   then E1.val = INT
                   else E1.val = ERROR
E -> E and E  if ((E2.val == BOOL) and (E3.val == BOOL))
              then E1.val = BOOL
                   else E1.val = ERROR
E -> E == E   if ((E2.val == E3.val) and
                 (E2.val != ERROR))
                   then E1.val = BOOL
                   else E1.val = ERROR
E -> true           E.val = BOOL
E -> false          E.val = BOOL
E -> n              E.val = INT
E -> ( E )          E1.val = E2.val
```

Type check example

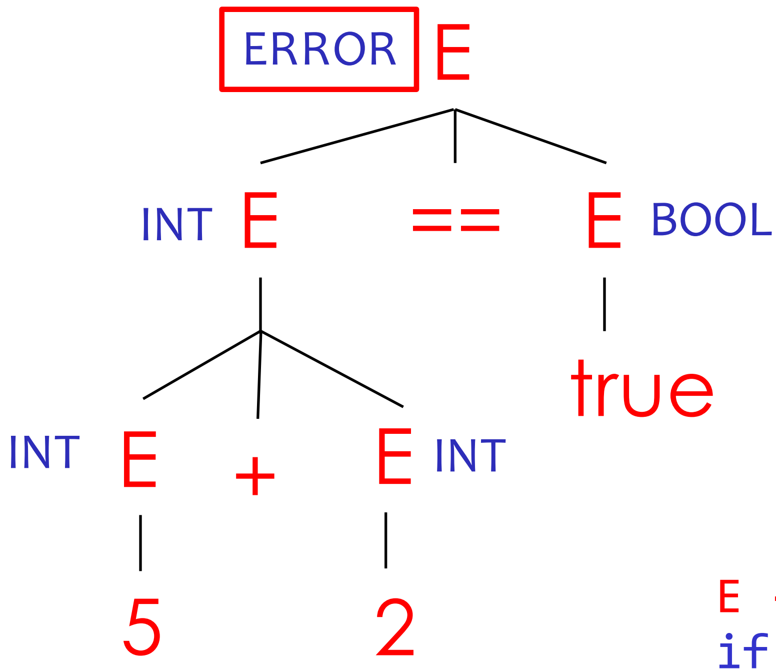


E -> n
E.val = INT

E -> E + E
if ((E_2.val == INT) and (E_3.val == INT))
then E_1.val = INT
else E_1.val = ERROR

E -> E == E
if ((E_2.val == E_3.val) and (E_2.val != ERROR))
then E_1.val = BOOL
else E_1.val = ERROR

Type check example



E -> true
E.val = BOOL

E -> E == E
if ((E_2.val == E_3.val) and (E_2.val != ERROR))
then E_1.val = BOOL
else E_1.val = ERROR

Another AG needing top-down pass

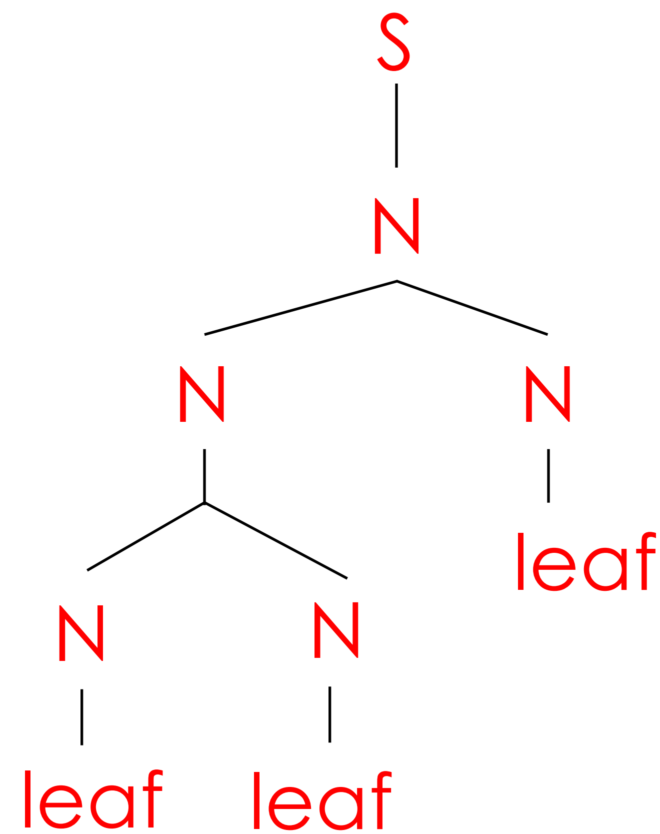
For each leaf node in parse tree, compute distance from the root:

$S \rightarrow N$

$N_1 \rightarrow$ leaf

$\quad | \; N_2 \; N_3$

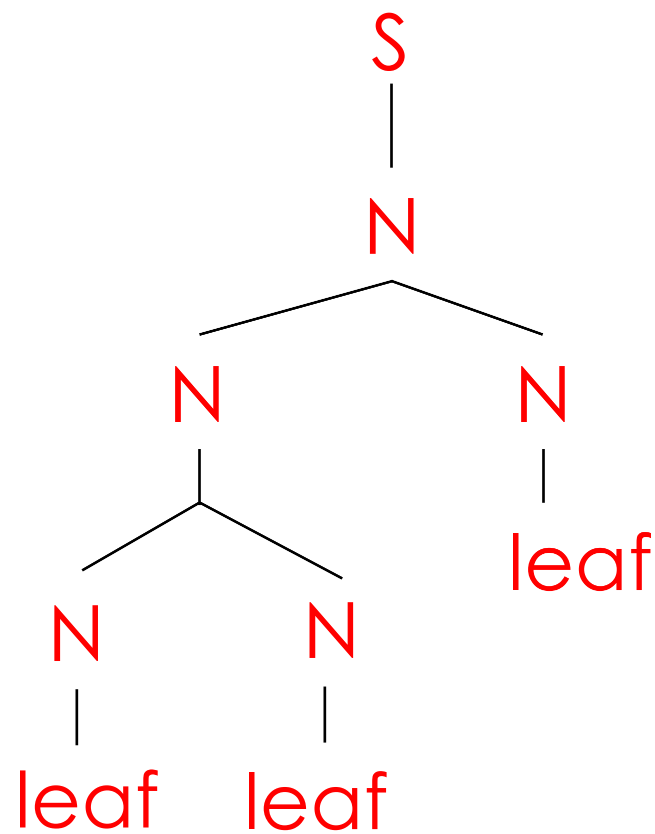
Let's add a `dist` attribute



Another AG needing top-down pass

For each leaf node in parse tree, compute distance from the root:

$S \rightarrow N$	$N.dist = 0$
$N_1 \rightarrow$ leaf	$leaf.dist = N_1.dist + 1$
$\quad \mid N_2 \; N_3$	$N_2.dist = N_1.dist + 1$
	$N_3.dist = N_1.dist + 1$



What kind of attribute is `dist`?

Syntax-directed translation

evaluate parse tree (to produce a value, AST, …)

Syntax-directed translation (SDT)

- Process of converting source language into target driven by actions associated with each rule

- We have seen various examples earlier with attribute grammars

- We have also seen this earlier with our bytecode compiler (PA2)

When is syntax directed translation performed?

Option 1: parse tree built explicitly during parsing

- after parsing, parse tree is traversed, rules are evaluated
- simpler, less efficient, but simpler; used in the 401 parser
- Necessary when the tree must be traversed multiple times

Option 2: parse tree never built

- rules evaluated during parsing on a conceptual parse tree
- more common in practice
- we'll see this strategy in HW3 (on recursive descent parser)

Let's construct an AST from parse tree

- We will use SDT for this purpose
- We will build the AST bottom up
- Each node will have a `val` attribute that stores the AST we have constructed for its descendants
- Steps:
  - Define the grammar
  - Define actions
  - Associate actions with production rules

An AG for AST building

$F \rightarrow$ int   F.val = new IntLitNode(int.value)

$F \rightarrow (E)$ F.val = E.val

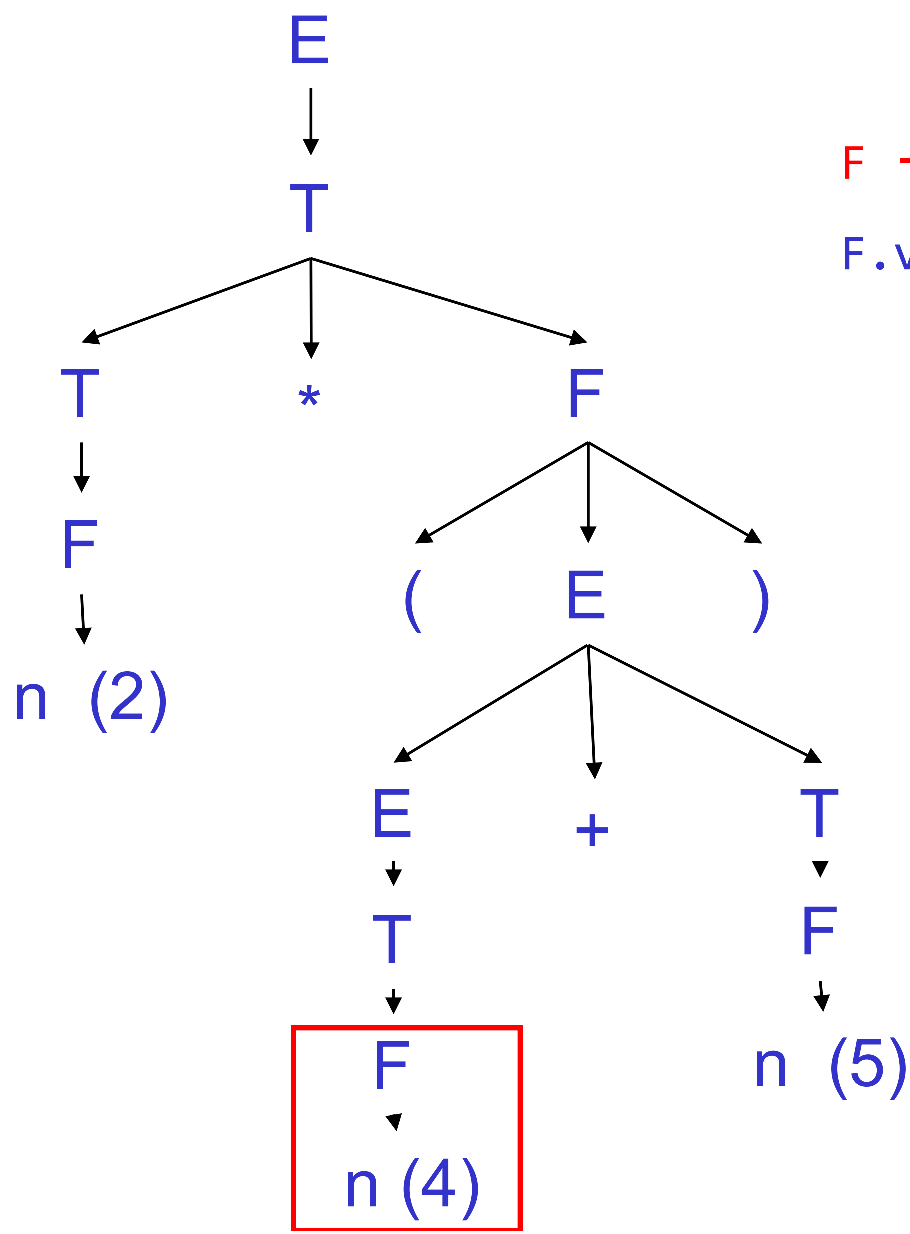
$T_1 \rightarrow T_2 * F$ T_1.val = new TimesNode(T_2.val, F.val)

$T \rightarrow F$ T.val = F.val

$E_1 \rightarrow E_2 + T$ E_1.val = new PlusNode(E_2.val, T.val)

$E \rightarrow T$ E.val = T.val

Example: build AST for 2 * (4 + 5)

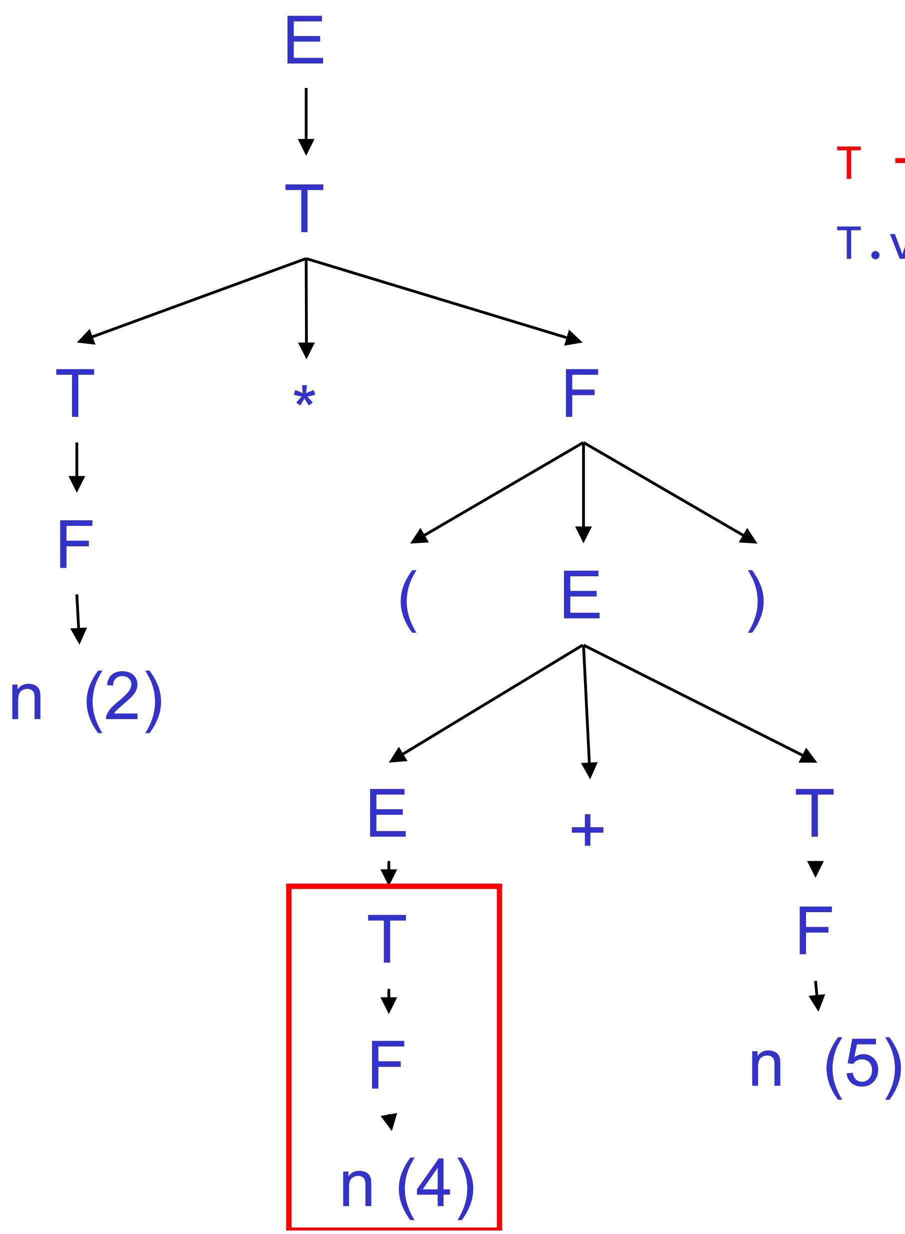



```
F → int

F.val = new IntLitNode(int.value)
```

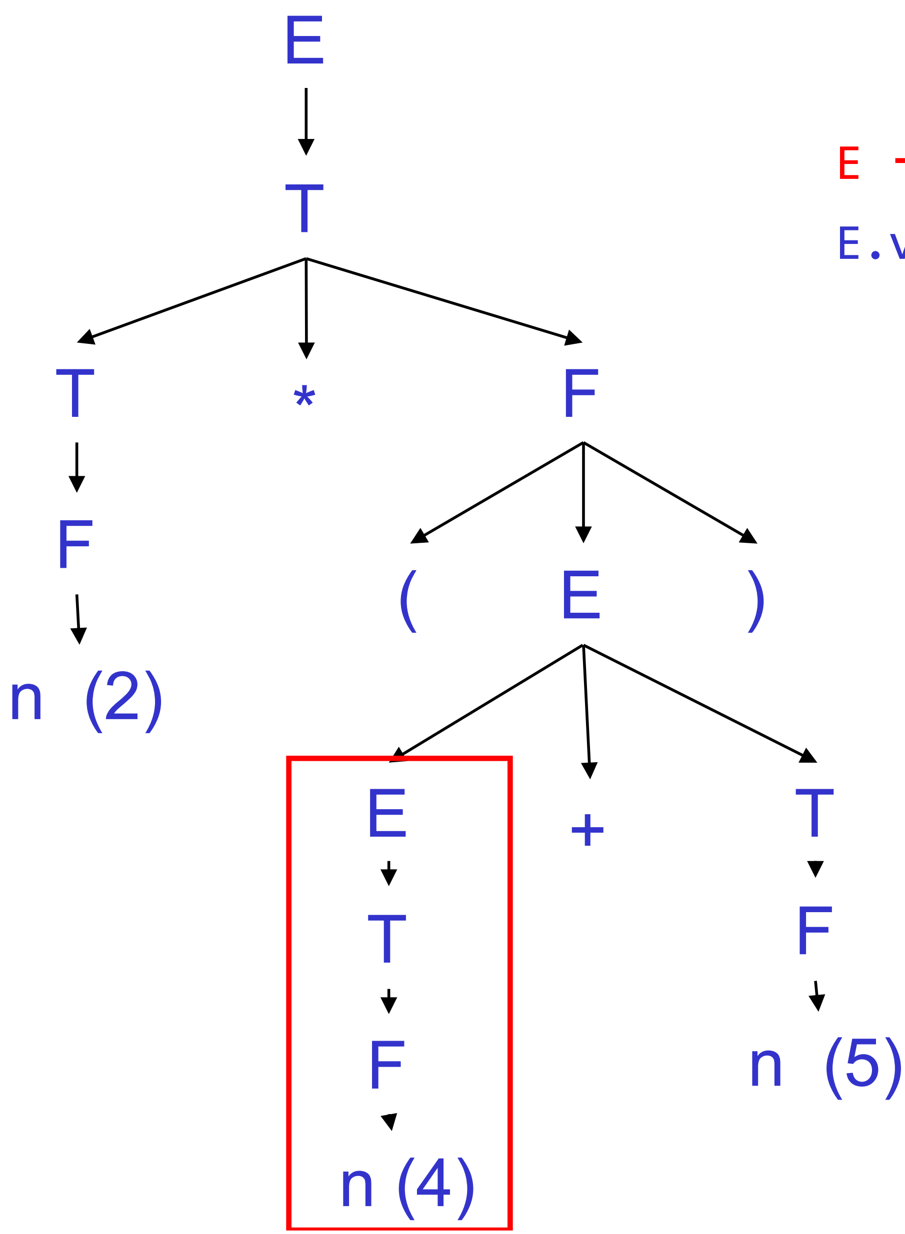
4

Example: build AST for 2 * (4 + 5)




```
T → F
T.val = F.val
```

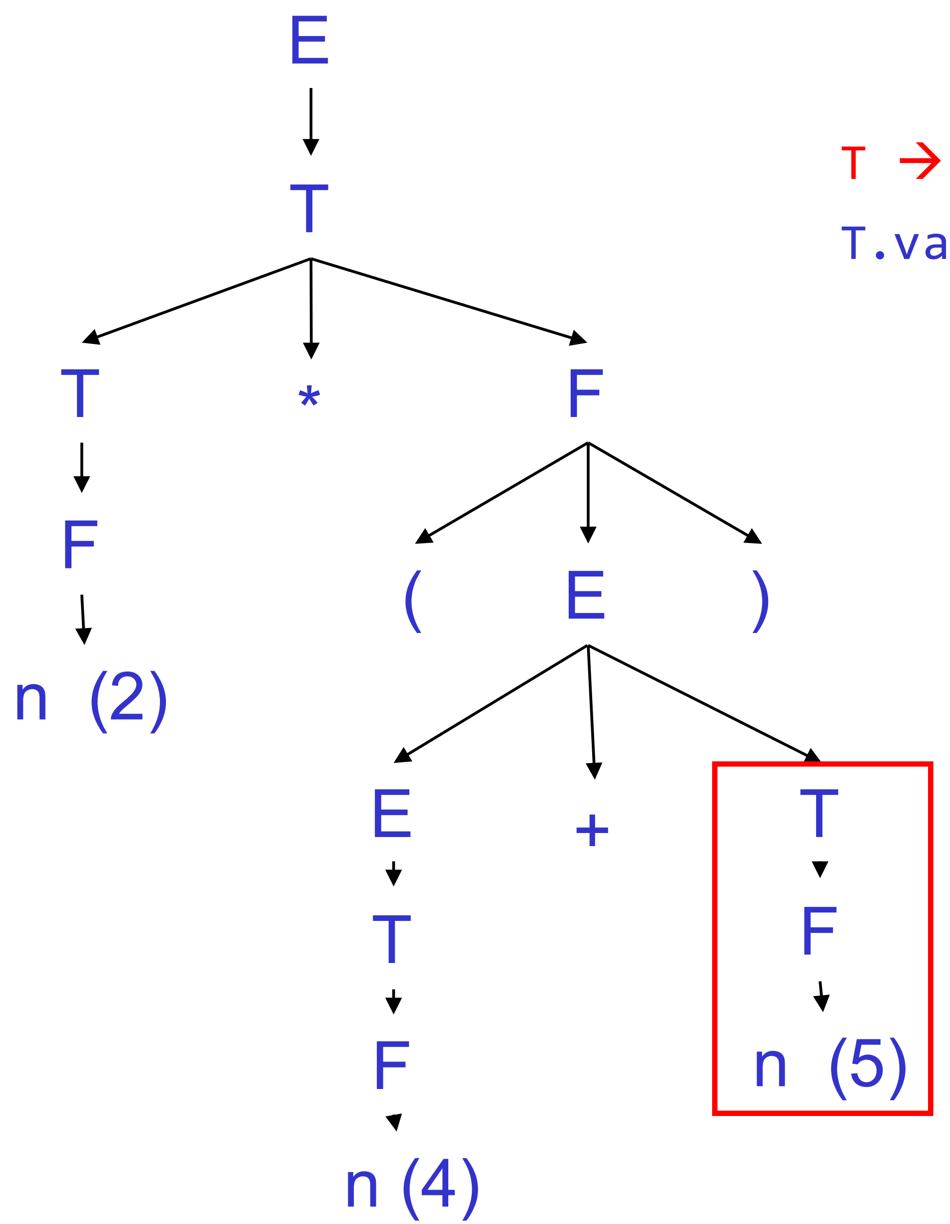
Example: build AST for 2 * (4 + 5)



E → T

E.val = T.val

Example: build AST for 2 * (4 + 5)

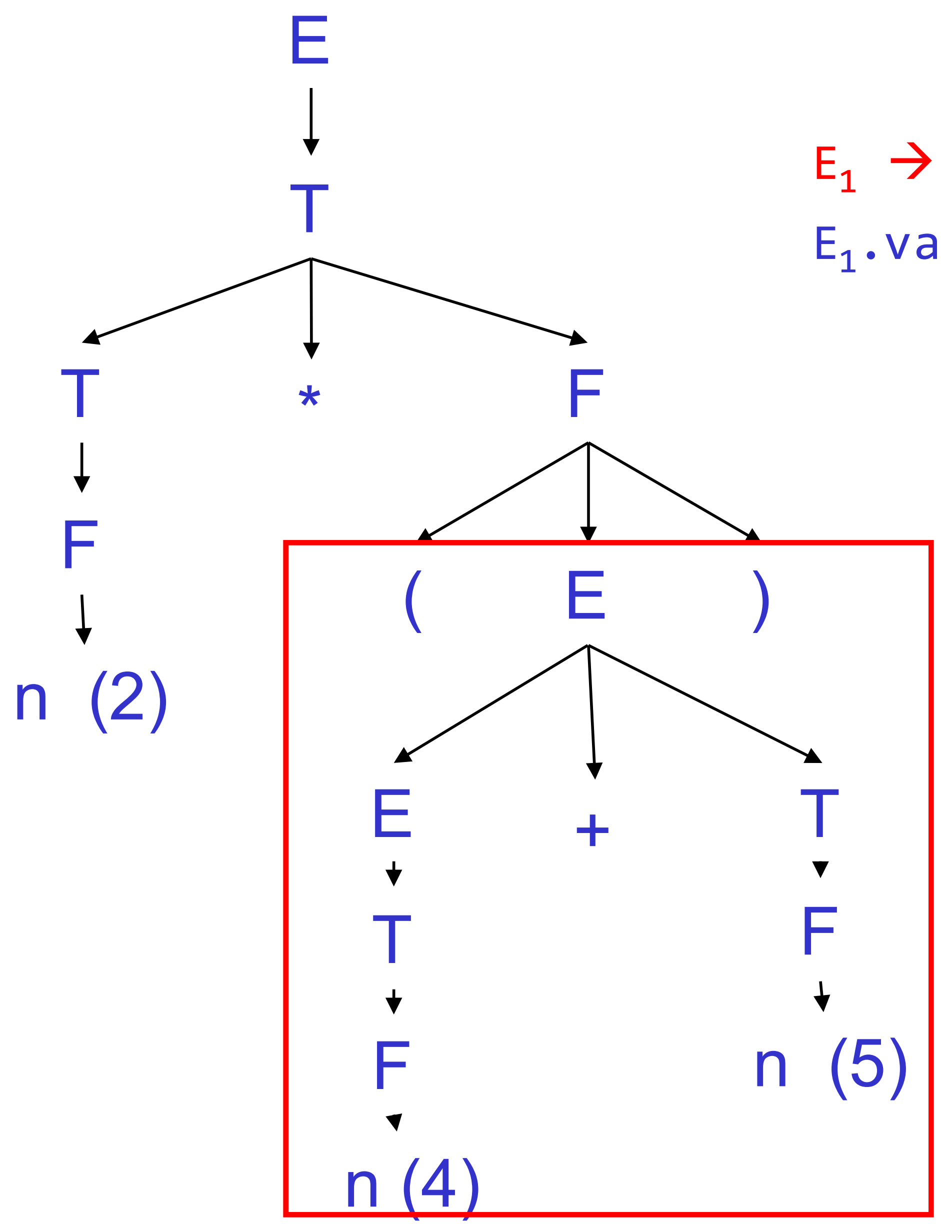


T → F

T.val = F.val

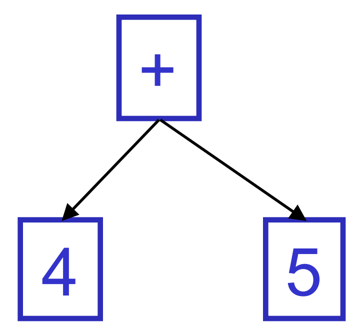
$\boxed{5}$

Example: build AST for 2 * (4 + 5)

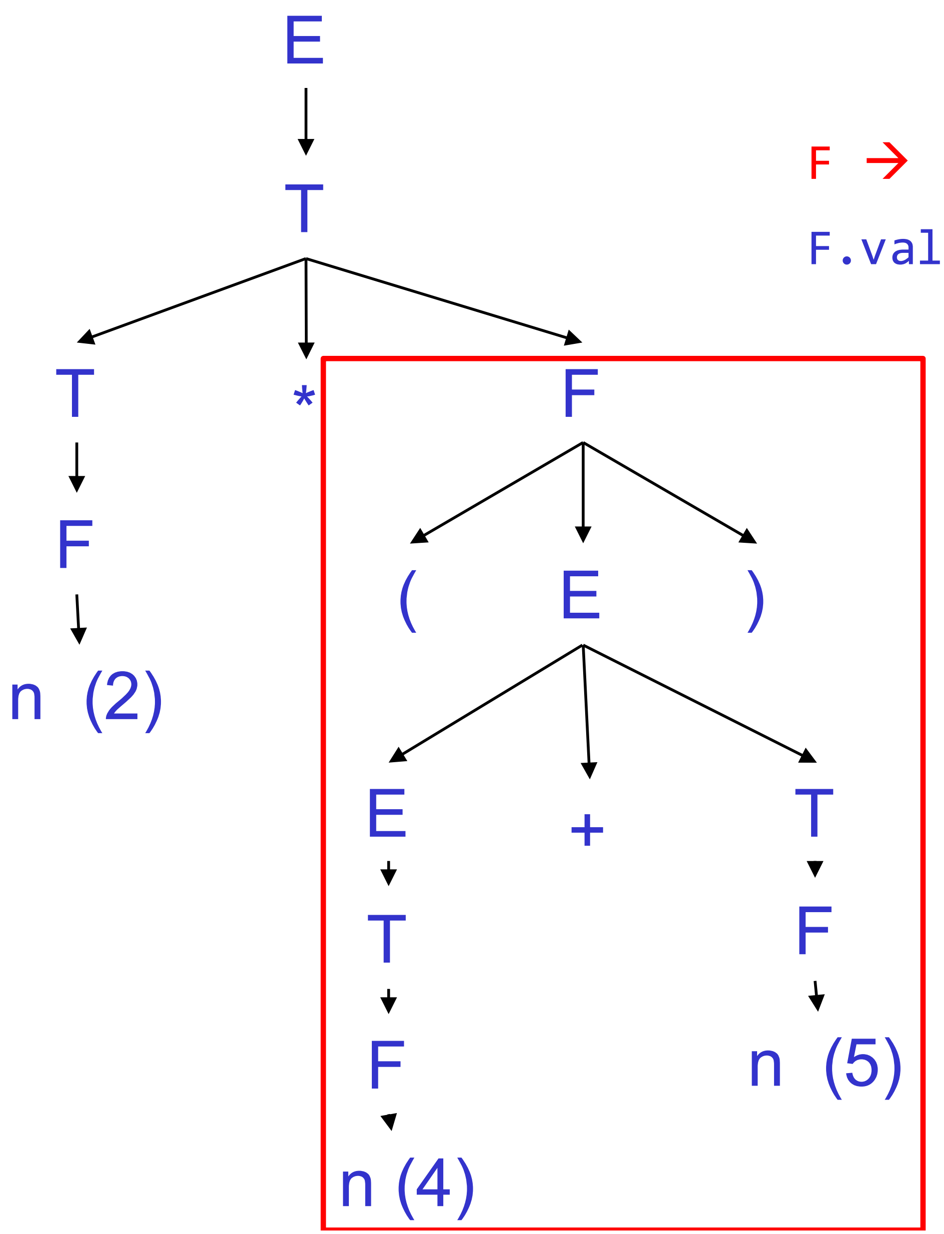


$E_1 \rightarrow E_2 + T$

E_1.val = new PlusNode(E_2.val, T.val)

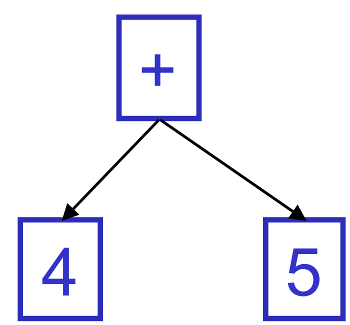


Example: build AST for 2 * (4 + 5)

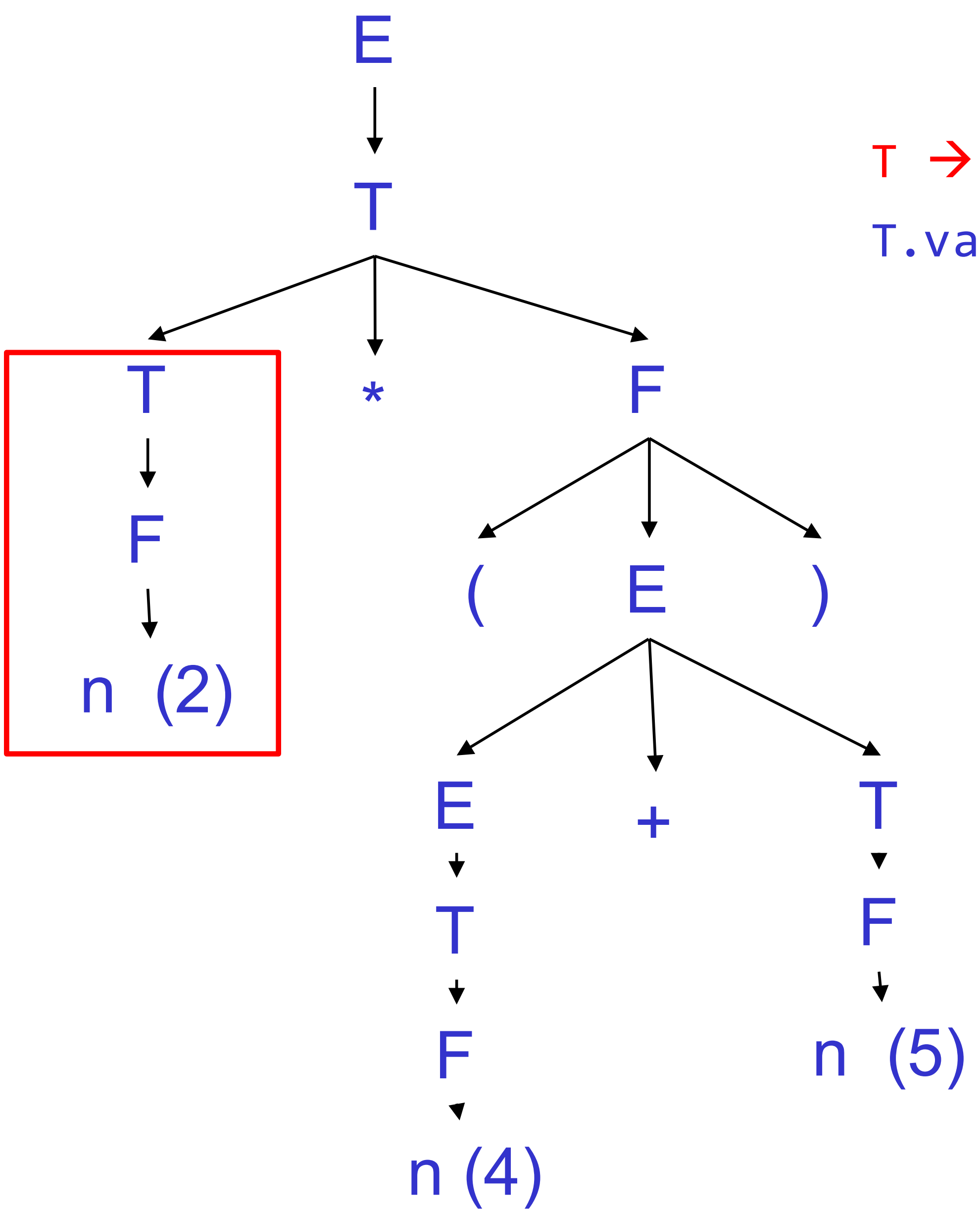


F → (E)

`F.val = E.val`



Example: build AST for 2 * (4 + 5)



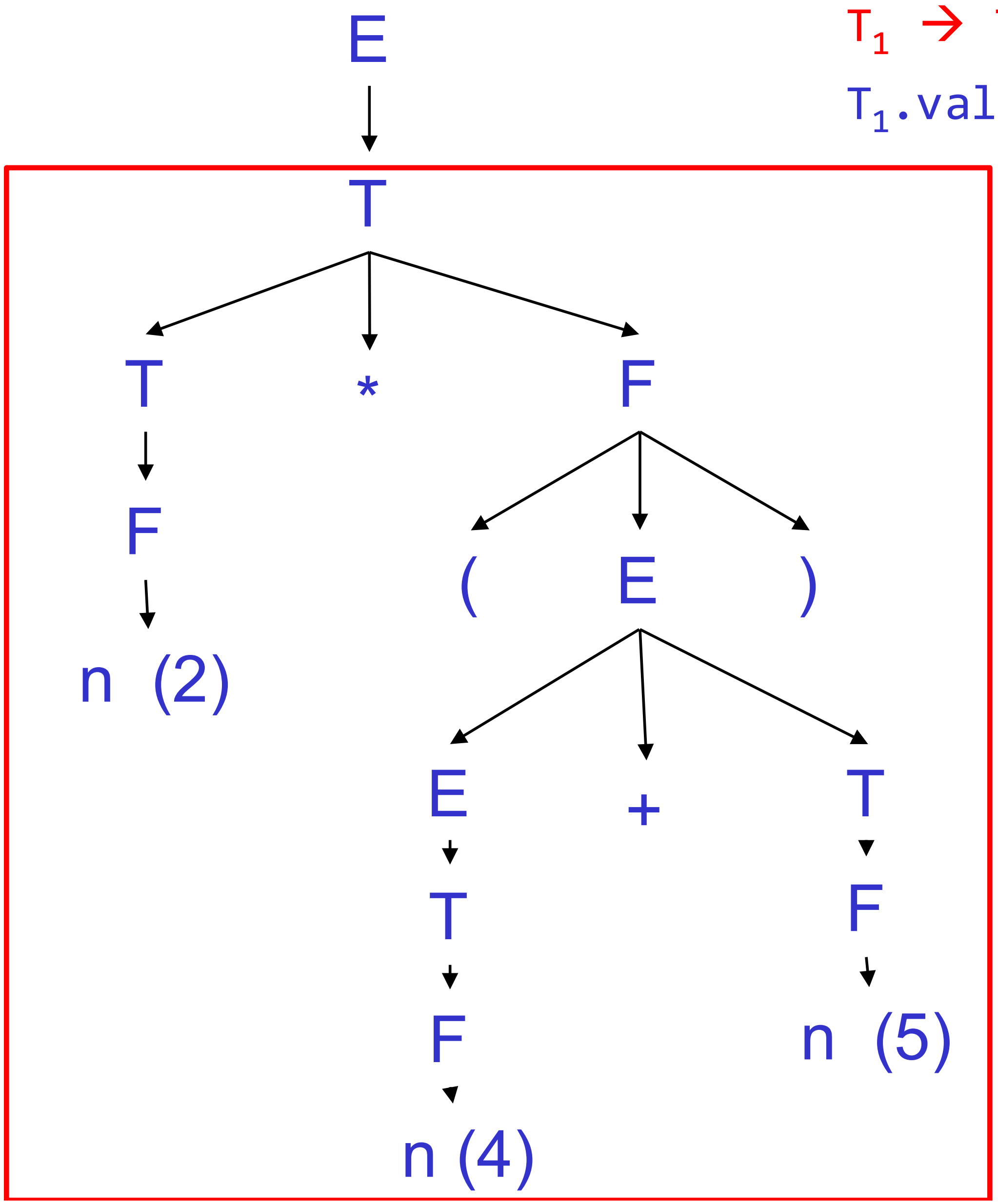
T → F

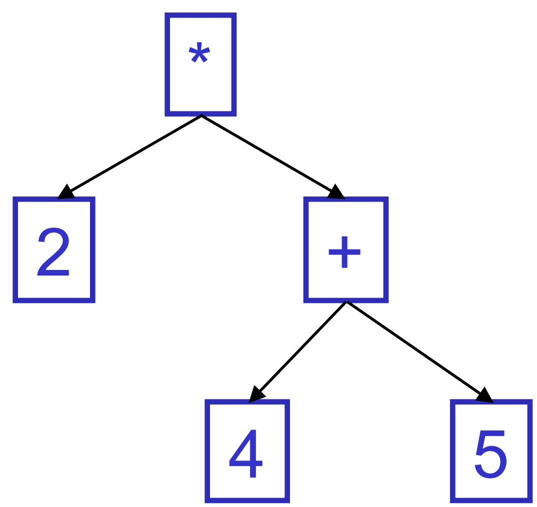
T.val = F.val

$\boxed{2}$

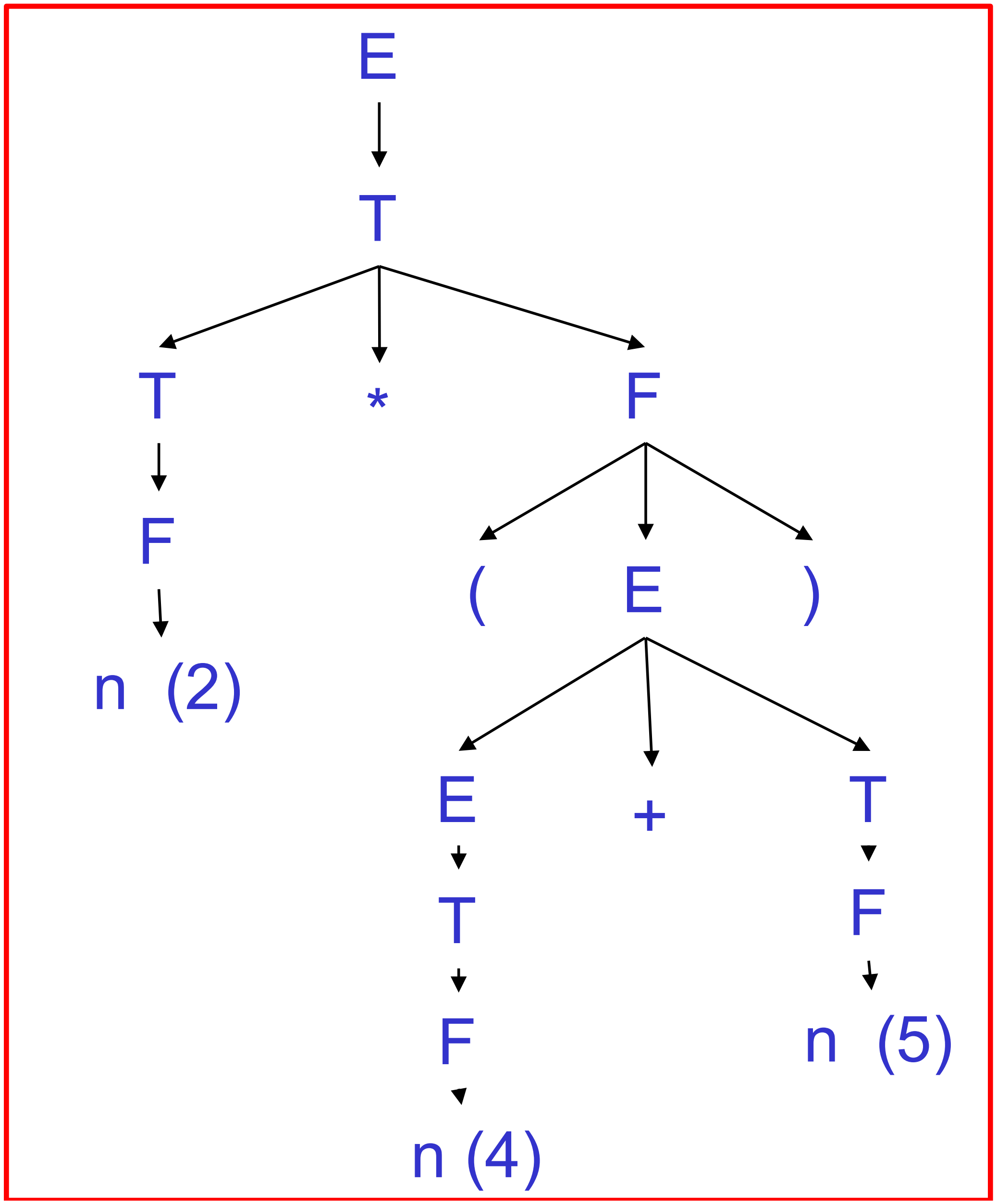
Example: build AST for 2 * (4 + 5)

```
T1 → T2 * F
T1.val = new TimesNode(T2.val, F.val)
```



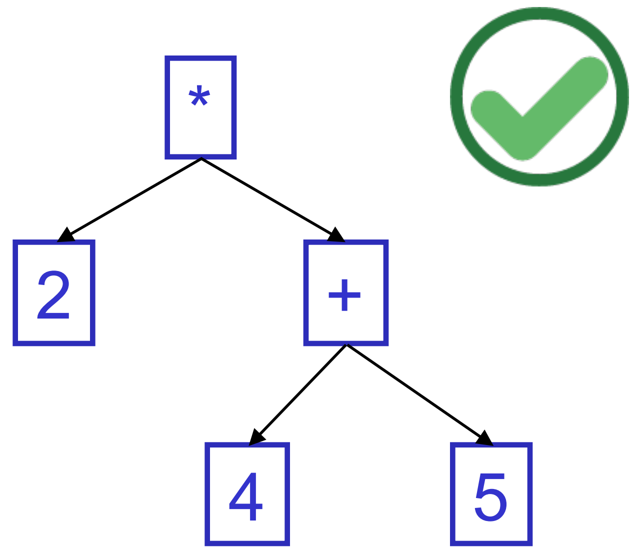


Example: build AST for 2 * (4 + 5)



E → T

E.val = T.val



“Multi-pass” Attribute Grammars

More than one traversal

When is a bottom-up pass insufficient to eval all attrs?

When an attr depends on an parent node attribute:

    top-down pass is needed

… or when it depends on a left sibling node attribute:

    in-order pass is needed

Pass = tree traversal

    bottom up (postorder), top-down (preorder), inorder

Summary

- Four elements of a grammar
- Parse trees from input source code
- Handling ambiguities by grammar rewriting
- Attribute grammars
- Constructing ASTs using Syntax-Driven Translation

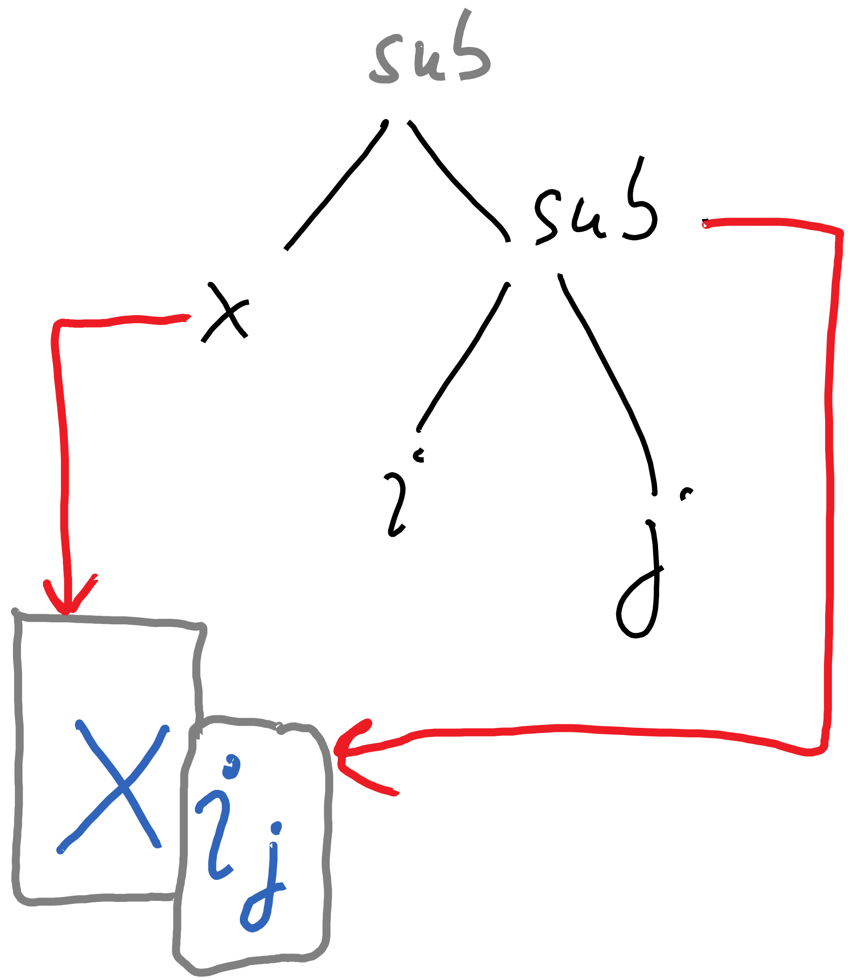
Multi-pass AG for simple math layout

Want to write

x **sub** i **sub** j

and obtain

x_{i_j}



is <u>sub</u> left- or right-associative?

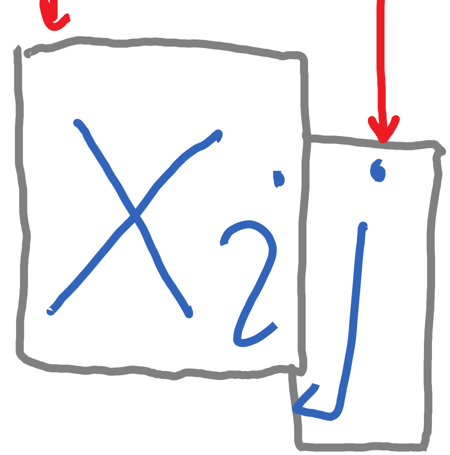
Multi-pass AG for Box layout

Want to write

(x **sub** i) **sub** j

and obtain

x_{ij}



The grammar

$B \rightarrow B_1\ B_2 \mid B_1\ \textbf{sub}\ B_2 \mid (\ B_1\ ) \mid \text{text}$

Attributes

What attributes do we need?

ps point size
ht height
dp depth

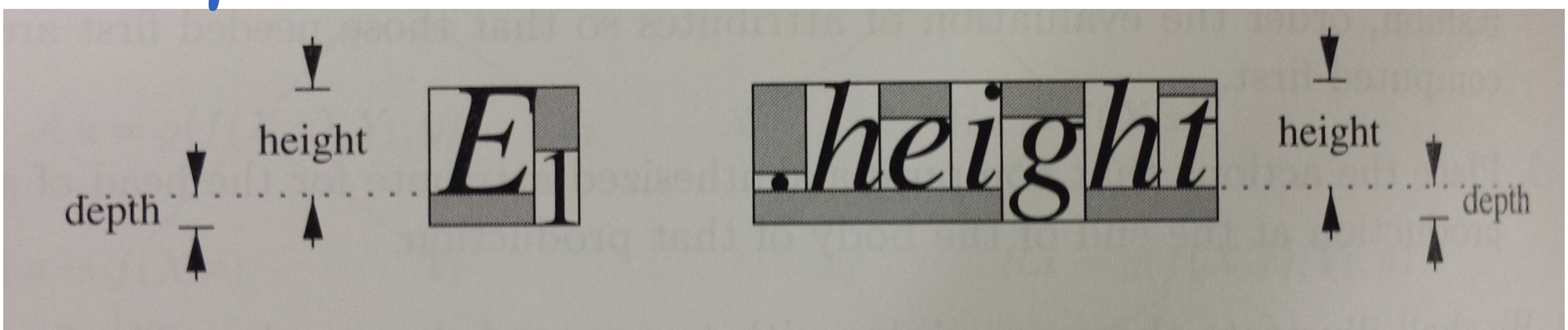


Figure 5.24: Constructing larger boxes from smaller ones

credits: example from Dragon Book

The AG (crucial rules only, see next slide for complete AG)

$S \rightarrow B$ B.ps = 10

$B \rightarrow B_1\ B_2$ B.ht = max (B_1.ht, B_2.ht)

same for dp

$B \rightarrow B_1$ **sub** B_2 B_2.ps = 0.7 * B.ps // shrink the font

B.ht = max (B_1.ht, $\underbrace{B_2\text{.ht} - 0.25 * \text{B.ps}}_{\text{shift } B_2 \text{ box down}}$)

$B \rightarrow (\ B_1\ )$

$B \rightarrow$ text   B.ht = getHt(B.ps, text.lexval)

how tall is the text in given ps?

The AG

	Production	Semantic Rules
1)	$S \to B$	$B.ps = 10$
2)	$B \to B_1 \ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \to B_1 \ \mathbf{sub} \ B_2$	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4)	$B \to (\ B_1 \ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \to \mathbf{text}$	$B.ht = getHt(B.ps, \mathbf{text}.lexval)$ $B.dp = getDp(B.ps, \mathbf{text}.lexval)$

The passes

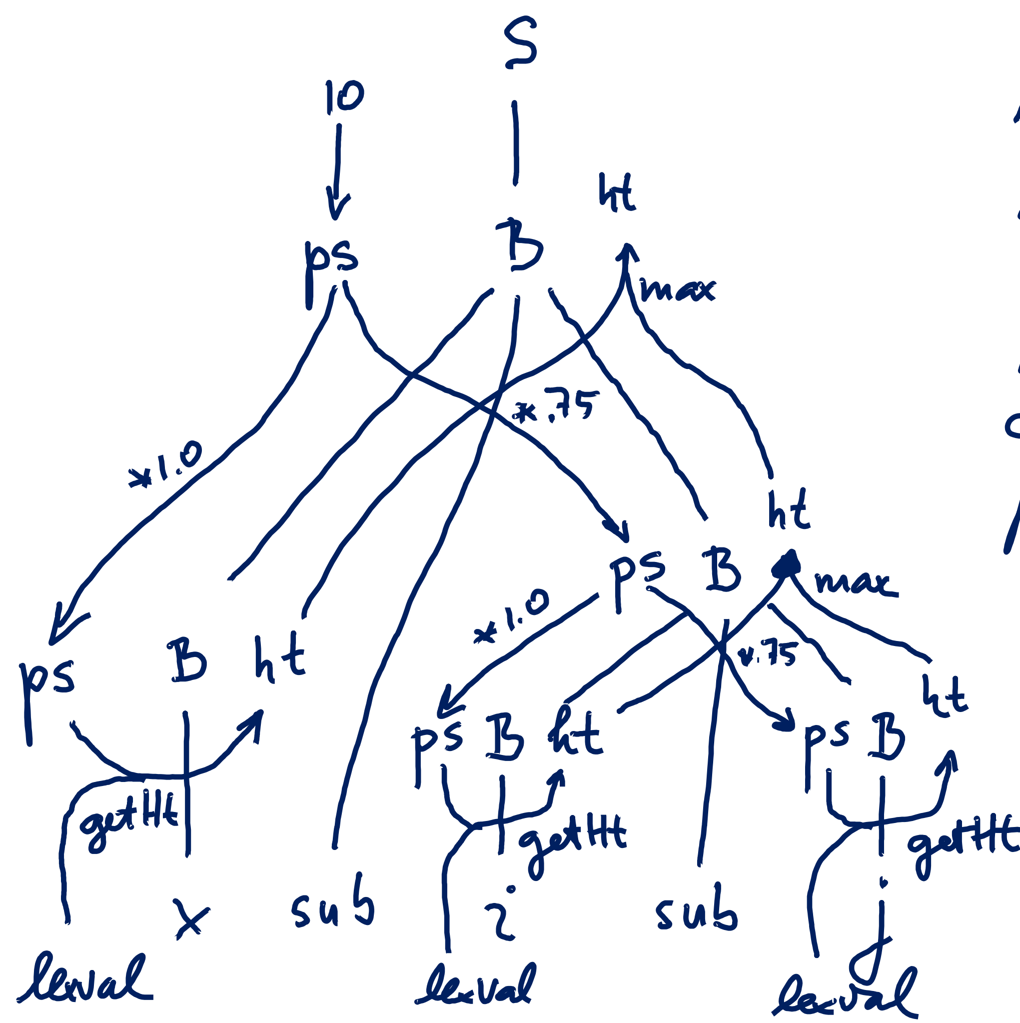
1st pass: top-down, computes ps
2nd pass: bottom-up, computes h, d

Example x **sub** i **sub** j

.ps

.height
.depth

Evaluation of `x sub i sub j`



1) top-down pass

1.5) leaf computation

2) bottom-up pass

note: leaf computation can be part of top-down pass or part of bottom-up pass.