**CS 160 Compilers**

# Lecture 10: Parsing Algorithms

Yu Feng
Fall 2021

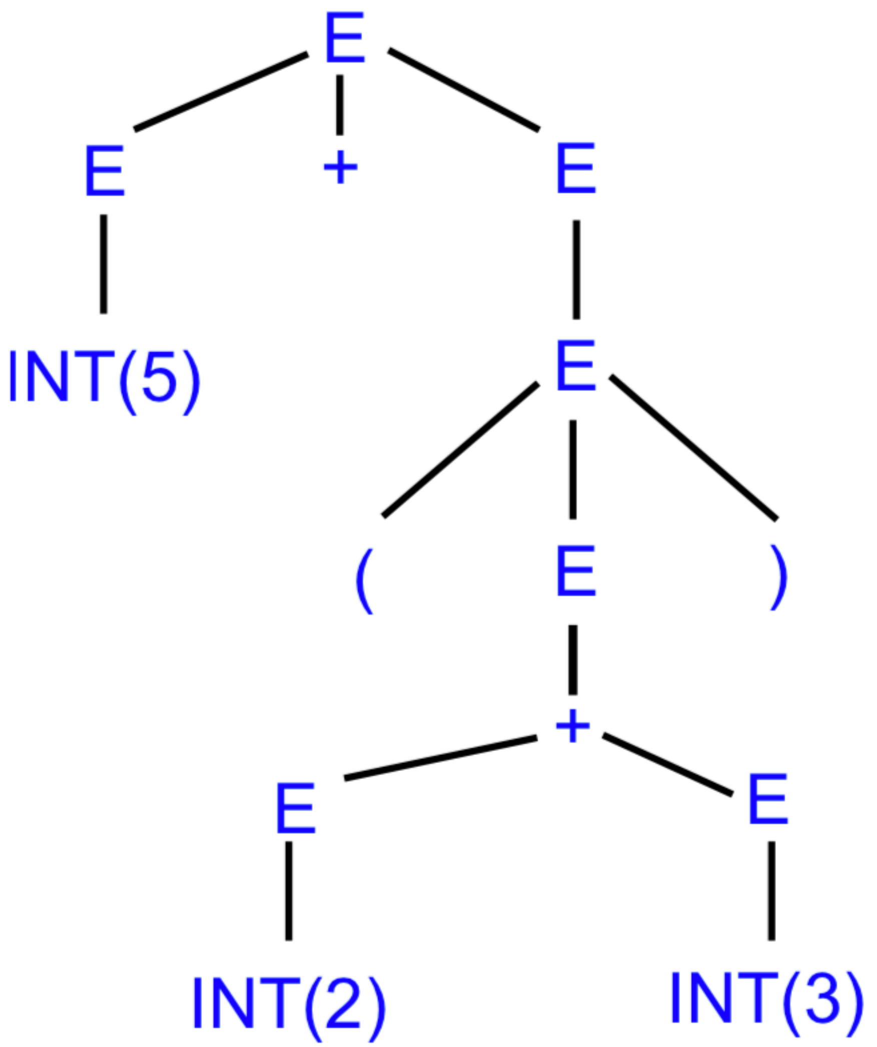
# Extend CFGs for program parsing

- CFGs describe the structure of a program
- But we also need this structure in form of a tree, not just a yes/no answer
- Insight: We do not need all program structure, only the relevant part
- We call this an *abstract syntax tree (AST)*

# ASTs

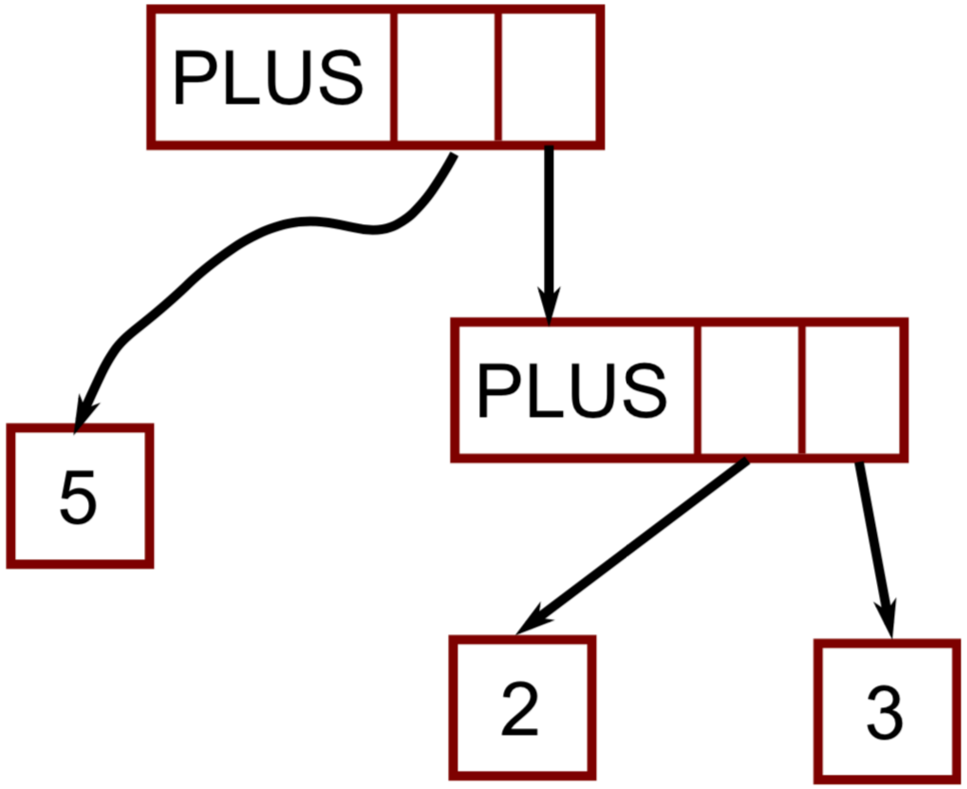
- Consider the grammar: E → int | (E) | E+E
- And the string: 5 + (2 + 3)
- After lexical analysis as string of tokens:
  - INT(5) ’+’ ’(’ INT(2) ’+’ INT(3) ’)
  - During parsing, we built a parse tree

# Example of parse tree



- Capture the nesting structure
- But too much information!
- Example: We do not care about the parentheses

# Example of abstract syntax tree



- Also captures the nesting structure
- But abstracts from the concrete syntax
- More compact and easier to use

# From CFG to AST

- Each grammar symbol has one attribute

Attribute Grammar

- For terminals (lexer tokens), the attribute is just the token
- Each production has a action computing its resulting attribute
- Written as: $X \rightarrow Y_1 \ldots Y_n \{\text{action}\}$

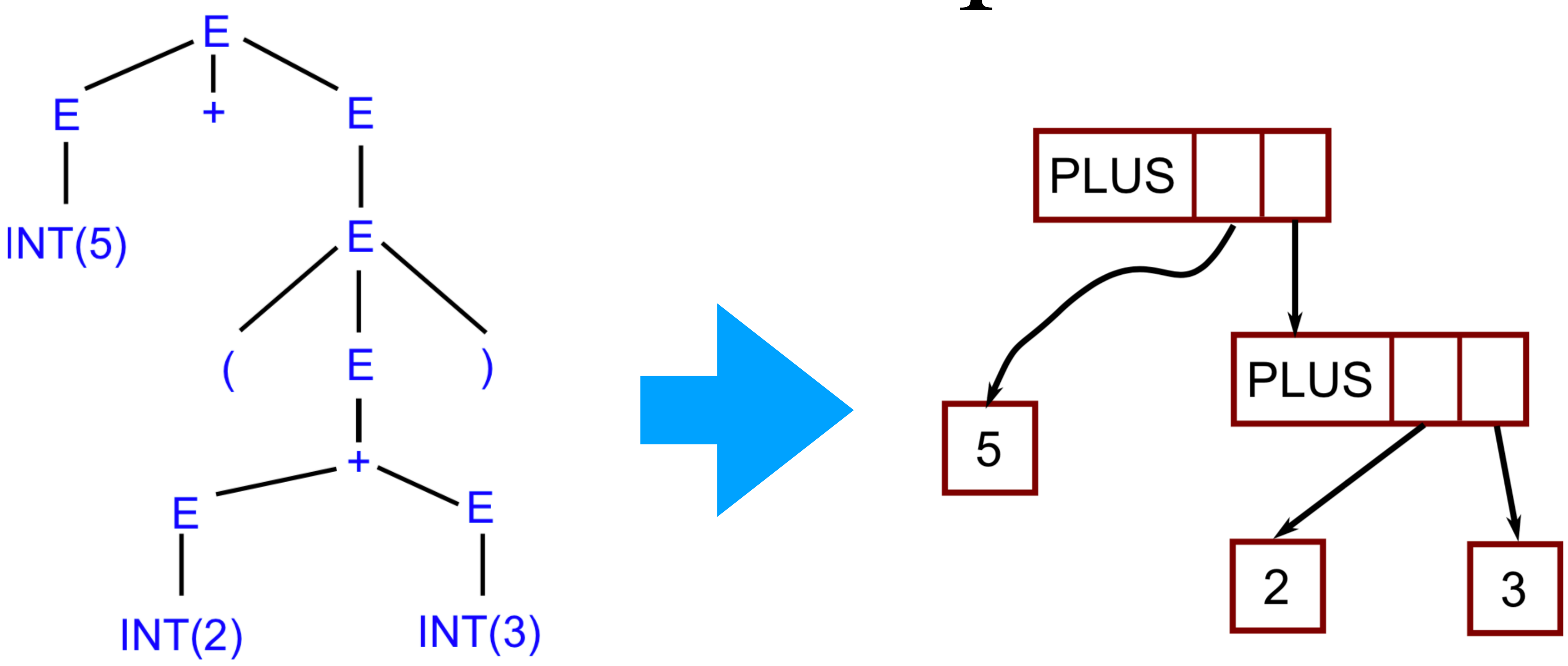
https://en.wikipedia.org/wiki/Attribute_grammar

# An example

- Consider again the grammar: E→int | (E) | E+E
- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side
- Recall: The value of each terminal is just its token
- Assume value of symbol S is given by S.val
- Grammar annotated with actions to compute the AST:

$$
\begin{array}{lcl}
E & \rightarrow & \mathsf{int} \;\; \{\mathtt{E.val} = \mathtt{int.val}\} \\
E & \rightarrow & E_1 + E_2 \;\; \{\mathtt{E.val} = \mathtt{makeAstPlus}(\mathtt{E}_1\mathtt{.val}, \mathtt{E}_2\mathtt{.val})\} \\
E & \rightarrow & (E') \;\; \{\mathtt{E.val} = \mathtt{E}'\mathtt{.val}\}
\end{array}
$$

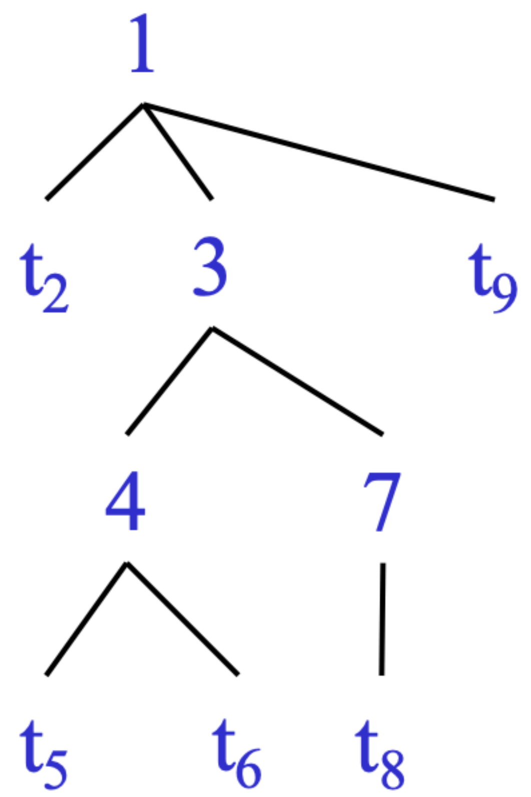
# An example



- You can think of semantic actions as defining a system of equations that describe the values of the let-hand sides in terms of values on the right-hand side

- Question: What order do we need to evaluate these equations to compute a solution?

# Top-Down parsing: the idea



- The parse tree is constructed
  - From the top
  - From left to right
- Terminals are seen in order of appearance in the token stream:
  - $t_2$ $t_5$ $t_6$ $t_8$ $t_9$

Recursive Descent Parsing

# Recursive descent parsing

- A Consider the grammar

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

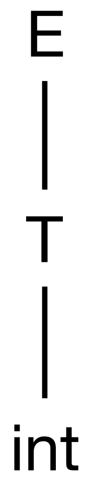
$$T \rightarrow \text{int} \mid \text{int} * T \mid ( E )$$

- Token stream is: ( $\text{int}_5$ )
- Start with top-level non-terminal E
- Try the rules for E in order

# Recursive descent parsing

E → T | T + E

T → int | int * T | ( E )



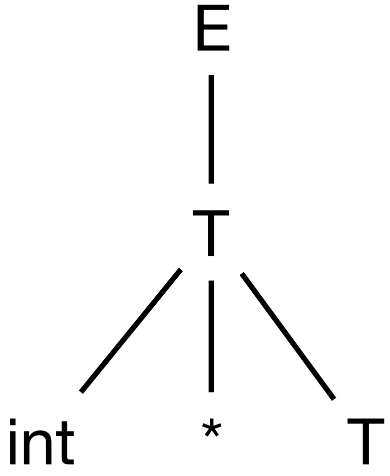
*Mismatch: int is not ( !*
*Backtrack …*

( int$_5$ )
↑

# Recursive descent parsing

E → T | T + E

T → int | int * T | ( E )



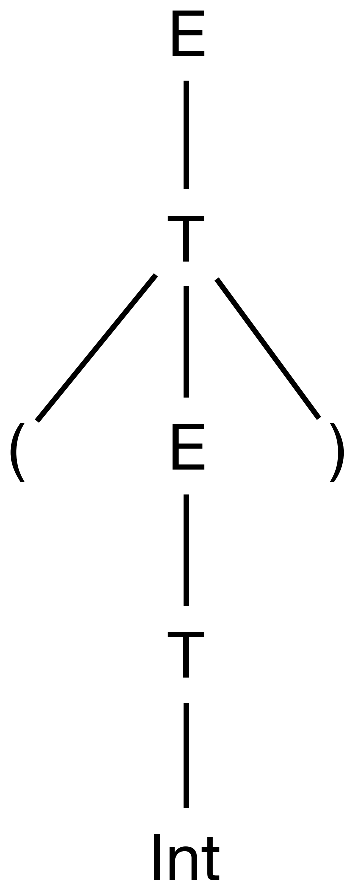
*Mismatch: int is not ( !*
*Backtrack …*

( int$_5$ )
↑

# Recursive descent parsing

E → T | T + E

T → int | int * T | ( E )



*Match: advance input*

*Accept: end of input*

( $\text{int}_5$ )

↑

# TODOs by next lecture

- Hw3 will be out.
- Come to the discussion session if you have questions