Introduction to Parsing

Lecture 8

Adapted from slides by G. Necula

Outline

- Limitations of regular languages
- Parser overview
- Context-free grammars (CFG's)
- Derivations

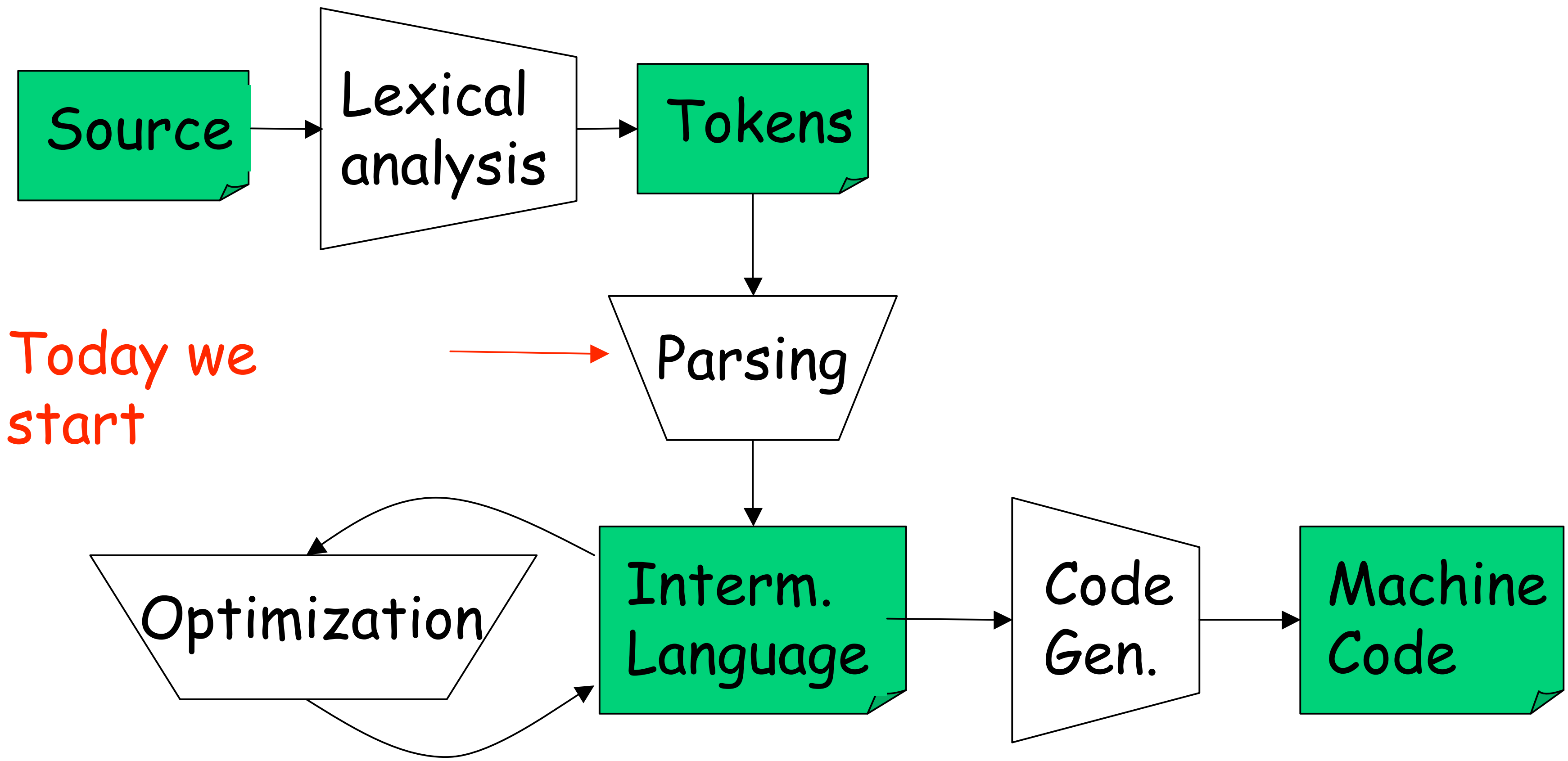
Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest formal languages widely used
  - Many applications
- We will also study context-free languages

Limitations of Regular Languages

- Intuition: A finite automaton that runs long enough must repeat states
- Finite automaton can't remember # of times it has visited a particular state
- Finite automaton has finite memory
  - Only enough to store in which state it is
  - Cannot count, except up to a finite limit
- E.g., language of balanced parentheses is not regular: $\{ (^i)^i \mid i \geq 0\}$

The Structure of a Compiler

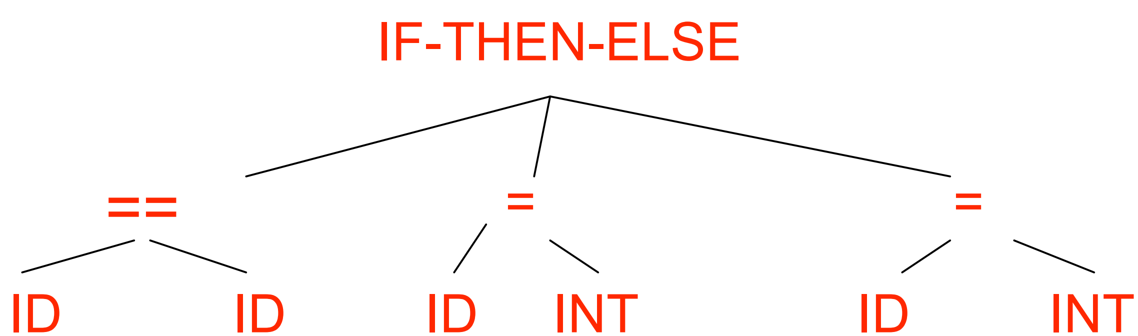


The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** abstract syntax tree of the program

Example

- Pyth: if x == y: z =1
  else: z = 2
- Parser input: IF ID == ID : ID = INT ↲ ELSE : ID = INT ↲
- Parser output (*abstract syntax tree)*:



Why A Tree?

- Each stage of the compiler has two purposes:
  - Detect and filter out some class of errors
  - Compute some new information or translate the representation of the program to make things easier for later stages
- Recursive structure of tree suits recursive structure of language definition
- With tree, later stages can easily find "the else clause", e.g., rather than having to scan through tokens to find it.

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Syntax tree

The Role of the Parser

- Not all sequences of tokens are programs . . .
- . . . Parser must distinguish between valid and invalid sequences of tokens

- We need
  - A language for describing valid sequences of tokens
  - A method for distinguishing valid from invalid sequences of tokens

Programming Language Structure

- Programming languages have recursive structure
- Consider the language of arithmetic expressions with integers, +, *, and ( )
- An expression is either:
  - an integer
  - an expression followed by "+" followed by expression
  - an expression followed by "*" followed by expression
  - a '(' followed by an expression followed by ')'
- int , int + int , (int + int) * int are expressions

Notation for Programming Languages

- An alternative notation:

$$E \rightarrow int$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$

- We can view these rules as rewrite rules
  - We start with E and replace occurrences of E with some right-hand side
- $E \rightarrow E * E \rightarrow (E) * E \rightarrow (E + E) * E \rightarrow \ldots$

  $\rightarrow (int + int) * int$

Observation

- All arithmetic expressions can be obtained by a sequence of replacements
- Any sequence of replacements forms a valid arithmetic expression
- This means that we cannot obtain

  (int))

  by any sequence of replacements. Why?
- This set of rules is a *context-free grammar*

Context-Free Grammars

- A CFG consists of
  - A set of *non-terminals* N
    - By convention, written with capital letter in these notes
  - A set of *terminals* T
    - By convention, either lower case names or punctuation
  - A *start symbol* S (a non-terminal)
  - A set of *productions*
- Assuming $E \in N$

$$E \to \varepsilon \qquad\qquad , \text{ or}$$

$$E \to Y_1\, Y_2 \ldots Y_n \qquad\qquad \text{where } Y_i \in N \cup T$$

Examples of CFGs

Simple arithmetic expressions:

$$E \rightarrow int$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$

- One non-terminal: E
- Several terminals: int, +, *, (,)
  - Called terminals because they are never replaced
- By convention the non-terminal for the first production is the start one

The Language of a CFG

Read productions as replacement rules:

$X \rightarrow Y_1 \ldots Y_n$

Means X can be replaced by $Y_1 \ldots Y_n$

$X \rightarrow \varepsilon$

Means X can be erased (replaced with empty string)

Key Idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any *non-terminal* X in the string by a right-hand side of some production

$$X \rightarrow Y_1 \ldots Y_n$$

3. Repeat (2) until there are only terminals in the string
4. The successive strings created in this way are called *sentential forms.*

The Language of a CFG (Cont.)

More formally, may write

$$X_1 \ldots X_{i-1} X_i X_{i+1} \ldots X_n \rightarrow X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

The Language of a CFG (Cont.)

Write

$$X_1 \ldots X_n \rightarrow^* y_1 \ldots y_m$$

if

$$X_1 \ldots X_n \rightarrow \ldots \rightarrow \ldots \rightarrow y_1 \ldots y_m$$

in 0 or more steps

The Language of a CFG

Let G be a context-free grammar with start symbol S. Then the language of G is:

$$L(G) = \{ a_1 \ldots a_n \mid S \rightarrow^* a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \}$$

Examples:

- $S \rightarrow 0$ also written as $S \rightarrow 0 \mid 1$

  $S \rightarrow 1$

  Generates the language { "0", "1" }
- What about $S \rightarrow 1\ A$

  $A \rightarrow 0 \mid 1$
- What about $S \rightarrow 1\ A$

  $A \rightarrow 0 \mid 1\ A$
- What about $S \rightarrow \varepsilon \mid (\ S\ )$

Pyth Example

A fragment of Pyth:

Compound → while Expr: Block
| if Expr: Block Elses

Elses → ε | else: Block | elif Expr: Block Elses

Block → Stmt_List | Suite

(Formal language papers use one-character non-terminals, but we don't have to!)

Notes

The idea of a CFG is a big step. But:

- Membership in a language is "yes" or "no"
  - we also need parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG's (e.g., bison)

More Notes

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar

  - Tools for regular languages (e.g., flex) are also sensitive to the form of the regular expression, but this is rarely a problem in practice

Derivations and Parse Trees

- A *derivation* is a sequence of sentential forms resulting from the application of a sequence of productions

$$S \rightarrow \ldots \rightarrow \ldots$$

- A derivation can be represented as a tree
  - Start symbol is the tree's root
  - For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1, \ldots, Y_n$ to node X

Derivation Example

- Grammar

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

- String

  int * int + int

Derivation Example (Cont.)

$$
\begin{aligned}
& E \\
\rightarrow\quad & E + E \\
\rightarrow\quad & E * E + E \\
\rightarrow\quad & \text{int} * E + E \\
\rightarrow\quad & \text{int} * \text{int} + E \\
\rightarrow\quad & \text{int} * \text{int} + \text{int}
\end{aligned}
$$

Derivation in Detail (1)

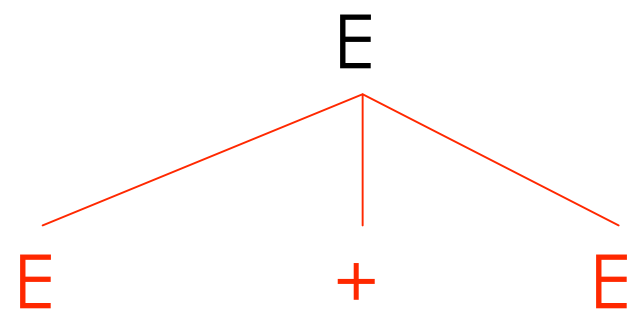
E

E

Derivation in Detail (2)

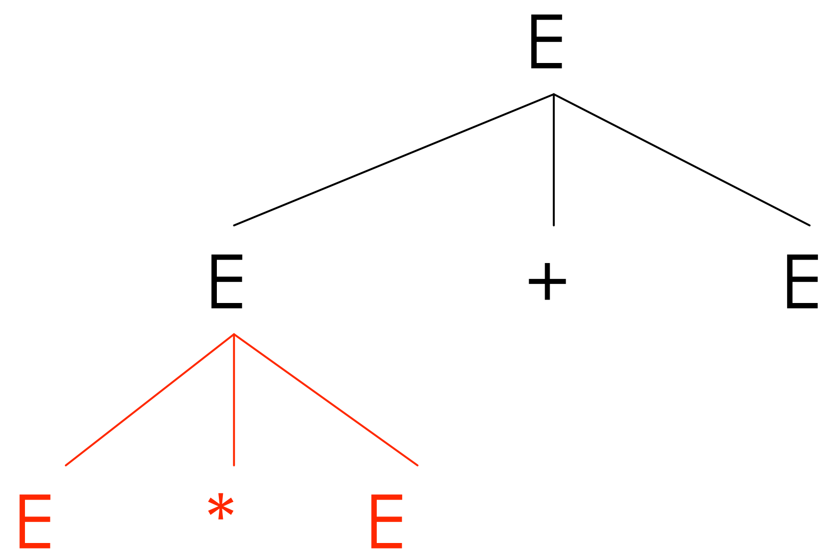
E

→ E + E



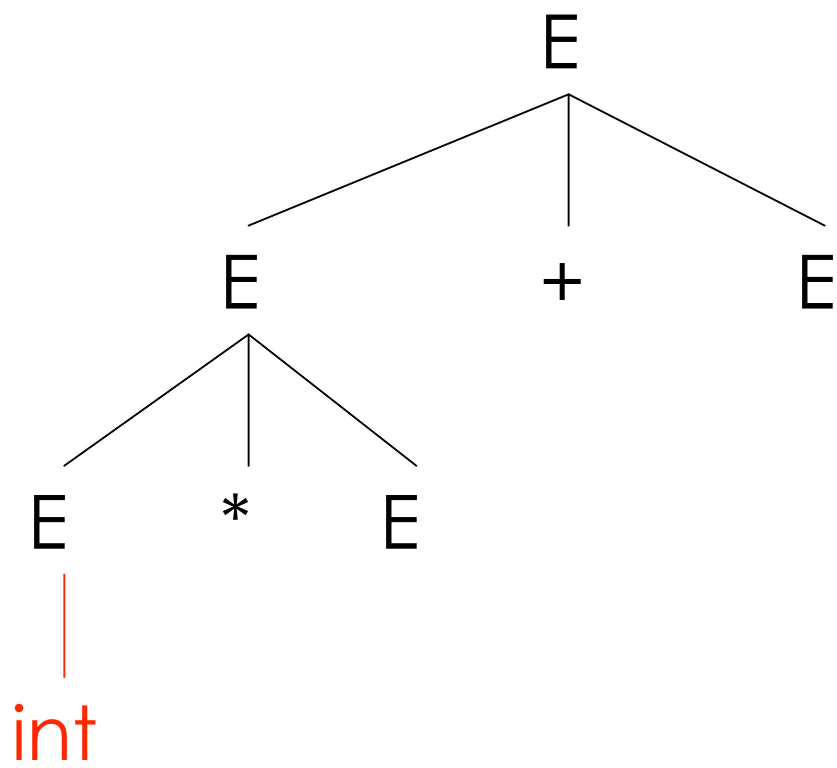
Derivation in Detail (3)

E
→ E + E
→ E * E + E



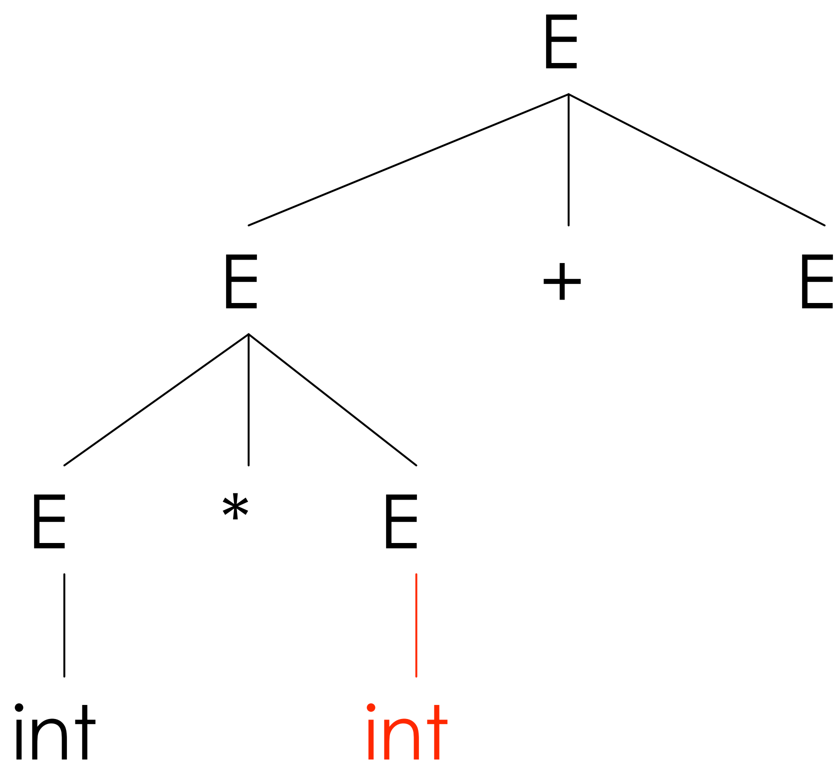
Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow\; & E + E \\
\rightarrow\; & E * E + E \\
\rightarrow\; & \text{int} * E + E
\end{aligned}
$$



Derivation in Detail (5)

$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E * E + E \\
\rightarrow\ & int * E + E \\
\rightarrow\ & int * int + E
\end{aligned}
$$



Derivation in Detail (6)

E

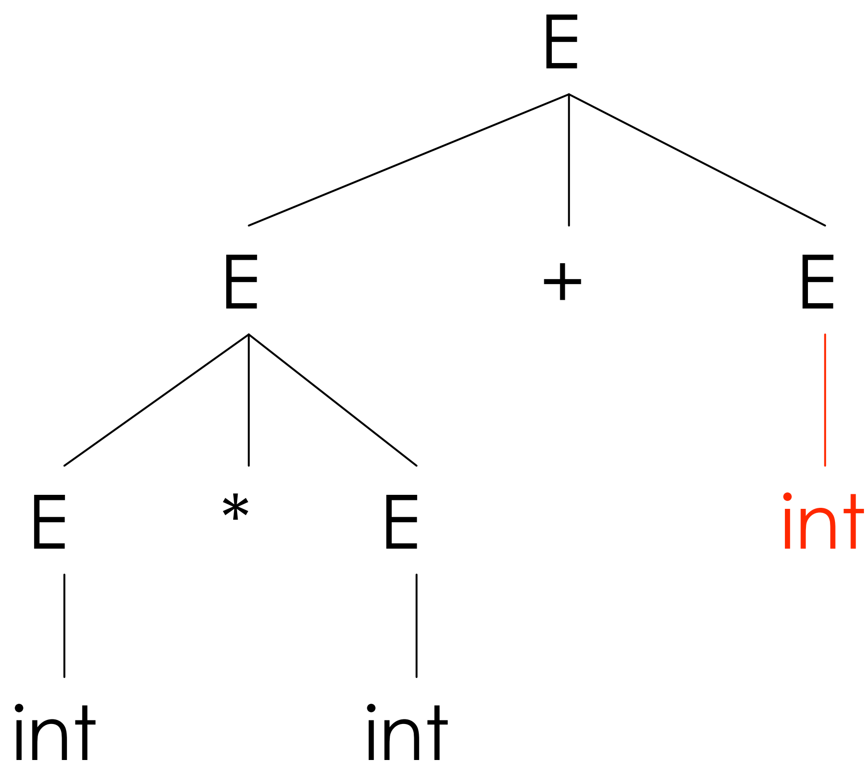
→ E + E

→ E * E + E

→ int * E + E

→ int * int + E

→ int * int + int



Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- A left-right traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not !
  - There may be multiple ways to match the input
  - Derivations (and parse trees) choose one

leftmost and Right-most Derivations

- The example was a *leftmost* derivation
  - At each step, replaced the leftmost non-terminal
- There is an equivalent notion of a *rightmost* derivation, shown here:

$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E + \text{int} \\
\rightarrow\ & E * E + \text{int} \\
\rightarrow\ & E * \text{int} + \text{int} \\
\rightarrow\ & \text{int} * \text{int} + \text{int}
\end{aligned}
$$

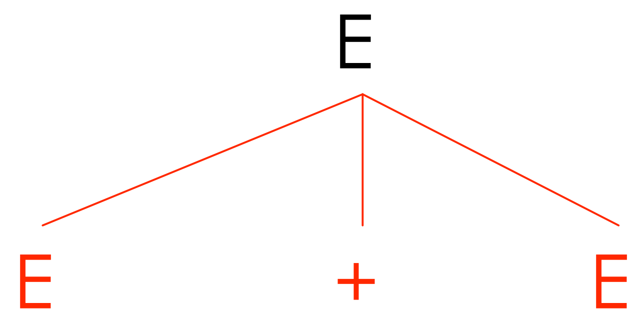
rightmost Derivation in Detail (1)

E

E

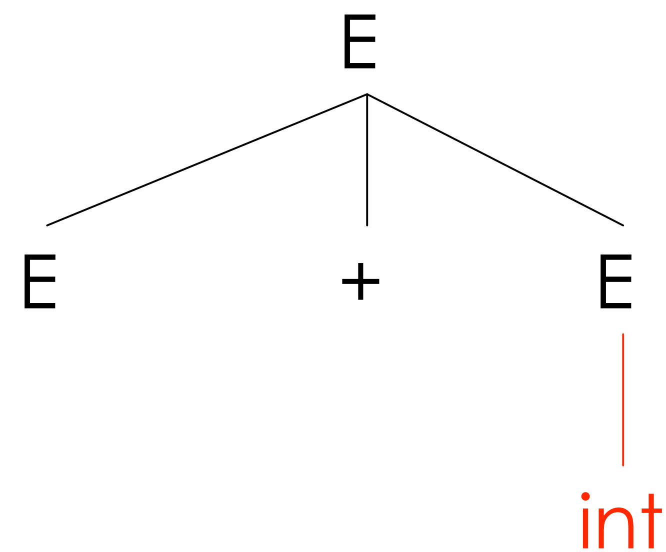
rightmost Derivation in Detail (2)

E
→ E + E



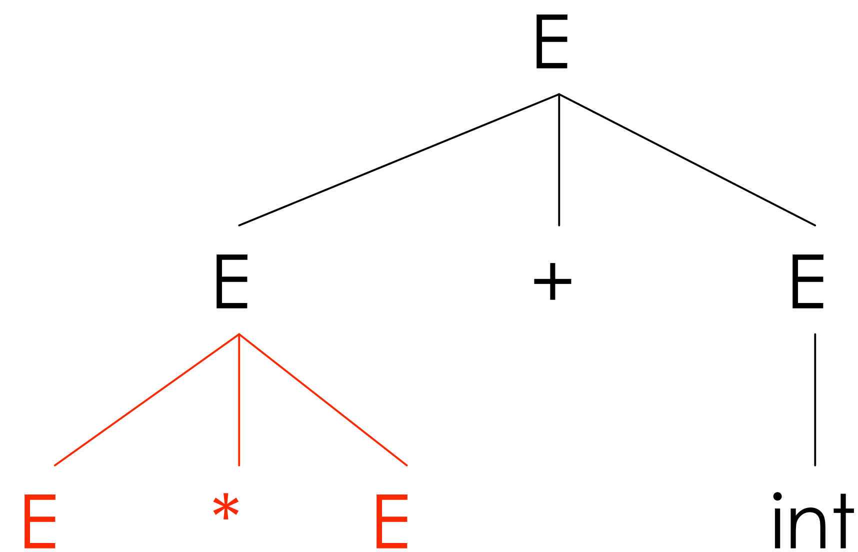
rightmost Derivation in Detail (3)

E
→ E + E
→ E + int



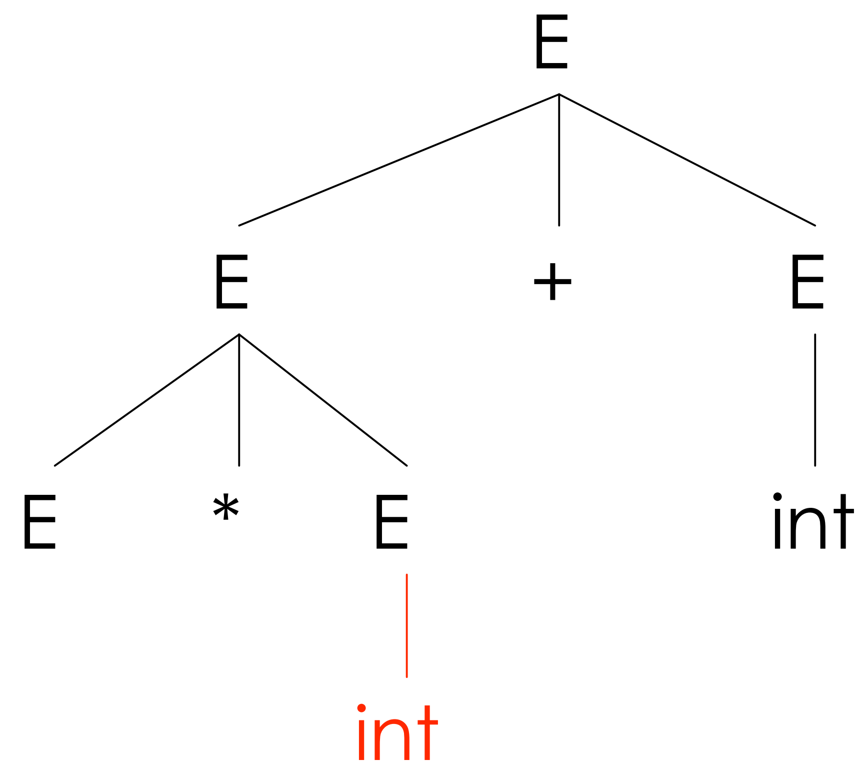
rightmost Derivation in Detail (4)

E
→ E + E
→ E + int
→ E * E + int



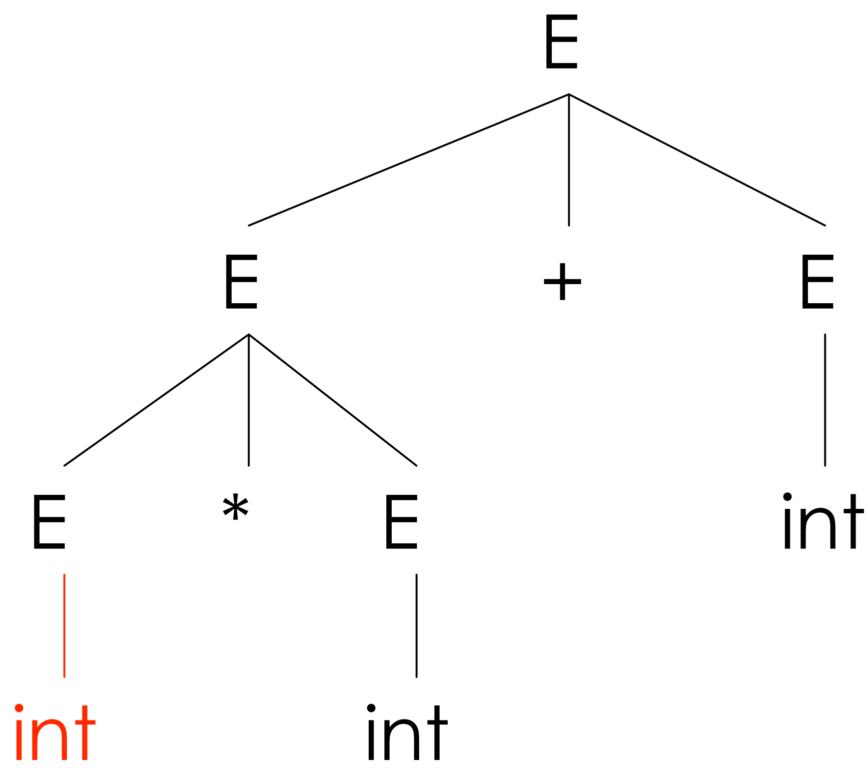
rightmost Derivation in Detail (5)

E
→ E + E
→ E + int
→ E * E + int
→ E * int + int



rightmost Derivation in Detail (6)

E
→ E + E
→ E + int
→ E * E + int
→ E * int + int
→ int * int + int



Aside: Canonical Derivations

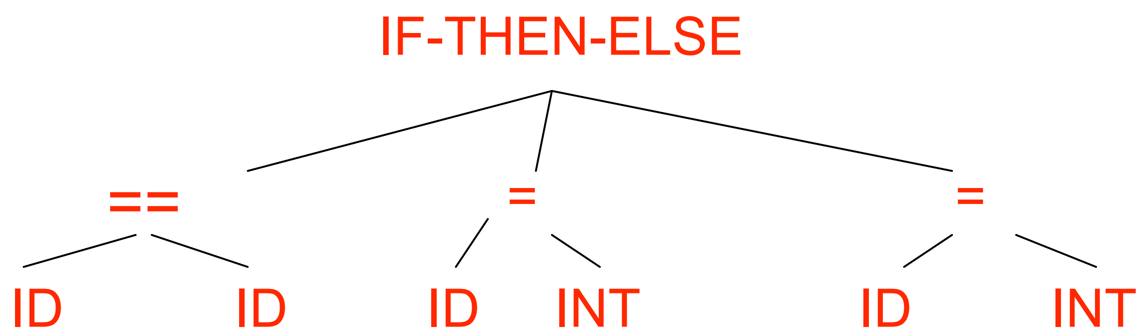
- Take a look at that last derivation *in reverse.*
- The active part (red) tends to move left to right.
- We call this a *reverse rightmost* or *canonical* derivation.
- Comes up in *bottom-up parsing.* We'll return to it in a couple of lectures.

Derivations and Parse Trees

- For each parse tree there is a leftmost and a rightmost derivation
- The difference is the order in which branches are added, not the structure of the tree.

Parse Trees and Abstract Syntax Trees

- The example we saw near the start:



was *not* a parse tree, but an *abstract syntax tree*

- Parse trees slavishly reflect the grammar.
- Abstract syntax trees more general, and abstract away from the grammar, cutting out detail that interferes with later stages.

Summary of Derivations

- We are not just interested in whether

$$s \in L(G)$$

  - We need a parse tree for s, and ultimately an abstract syntax tree.
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- leftmost and rightmost derivations are important in parser implementation