Introduction to Parsing

Lecture 4

Administrivia

- Programming Assignment 2 is out this week
  - Due October 1st
  - Work in teams begins
- Required Readings
  - Lex Manual
  - Red Dragon Book Chapter 4

Outline

- Regular languages revisited
- 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: $\{ (^i)^i \mid i$ ¸ $0\}$

The Functionality of the Parser

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

Example

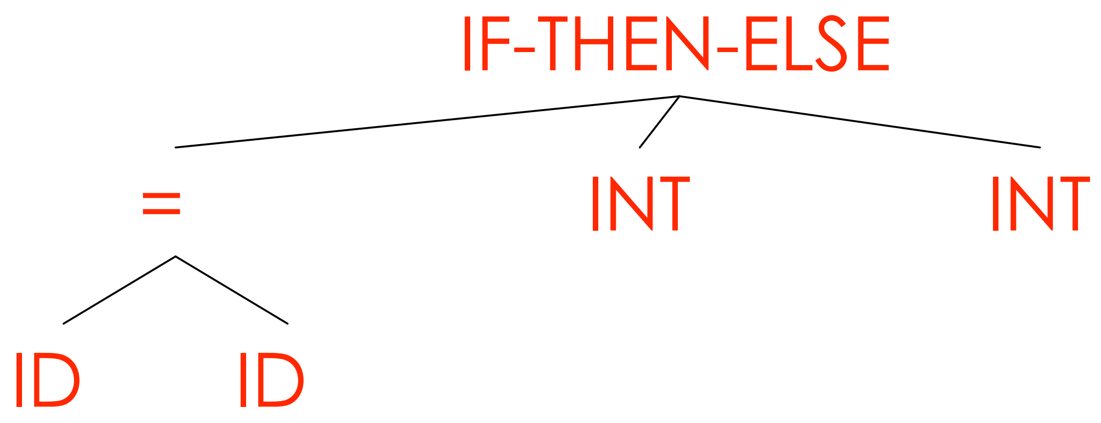
- Cool

  if x = y then 1 else 2 fi

- Parser input

  IF ID = ID THEN INT ELSE INT FI

- Parser output



Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse 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*

Context-Free Grammars

- Programming language constructs have recursive structure
- An EXPR is

  if EXPR then EXPR else EXPR fi , or

  while EXPR loop EXPR pool , or

  …
- Context-free grammars are a natural notation for this recursive structure

CFGs (Cont.)

- A CFG consists of
  - A set of *terminals* T
  - A set of *non-terminals* N
  - A *start symbol* S (a non-terminal)
  - A set of *productions*

Assuming $X \in N$

$X \Rightarrow \varepsilon$, or

$X \Rightarrow Y_1\, Y_2 \ldots Y_n$ where $Y_i \in (N \cup T)$

Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

Examples of CFGs

A fragment of Cool:

```
EXPR  →  if EXPR then EXPR else EXPR fi
      |  while EXPR loop EXPR pool
      |  id
```

Examples of CFGs (cont.)

Simple arithmetic expressions:

$$
\begin{array}{lll}
E & \rightarrow & E * E \\
  & | & E + E \\
  & | & (E) \\
  & | & \mathrm{id}
\end{array}
$$

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 no non-terminals in the string

The Language of a CFG (Cont.)

More formally, write

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

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:

$\{ a_1 \dots a_n \mid S \Rightarrow^* a_1 \dots a_n$ and every a_i is a terminal $\}$

Terminals

- Terminals are called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals ought to be tokens of the language

Examples

L(G) is the language of CFG G

Strings of balanced parentheses $\left\{ (^i\,)^i \mid i \geq 0 \right\}$

Two grammars:

$$\begin{array}{lll} S & \rightarrow & (S) \\ S & \rightarrow & \varepsilon \end{array} \qquad \textit{OR} \qquad \begin{array}{lll} S & \rightarrow & (S) \\ & \mid & \varepsilon \end{array}$$

Cool Example

A fragment of COOL:

$$
\begin{array}{lcl}
\text{EXPR} & \rightarrow & \text{if EXPR then EXPR else EXPR fi} \\
 & | & \text{while EXPR loop EXPR pool} \\
 & | & \text{id}
\end{array}
$$

Cool Example (Cont.)

Some elements of the language

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id

if if id then id else id fi then id else id fi

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow E+E \mid E * E \mid (E) \mid \mathrm{id}$$

Some elements of the language:

id	id + id
(id)	id * id
(id) * id	id * (id)

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

  - Note: 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 productions

$$S \Rightarrow \ldots \Rightarrow \ldots$$

A derivation can be drawn 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

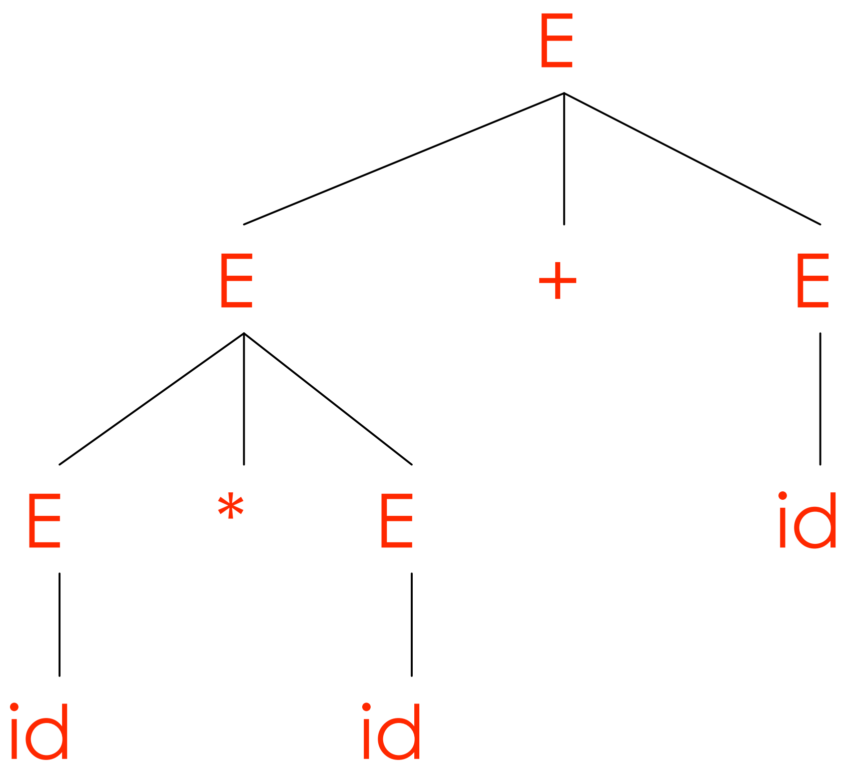
$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example (Cont.)

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

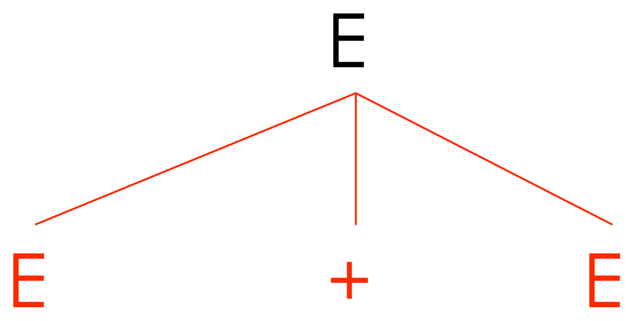


Derivation in Detail (1)

E

E

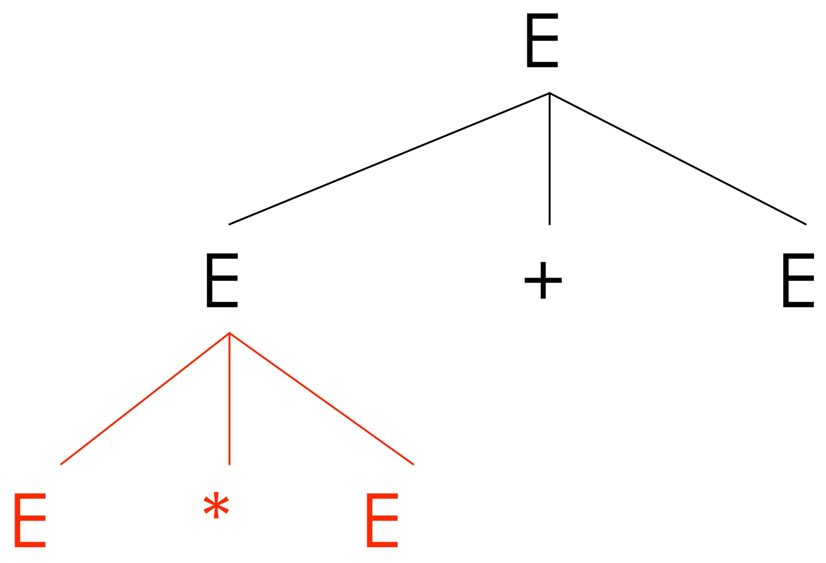
Derivation in Detail (2)



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

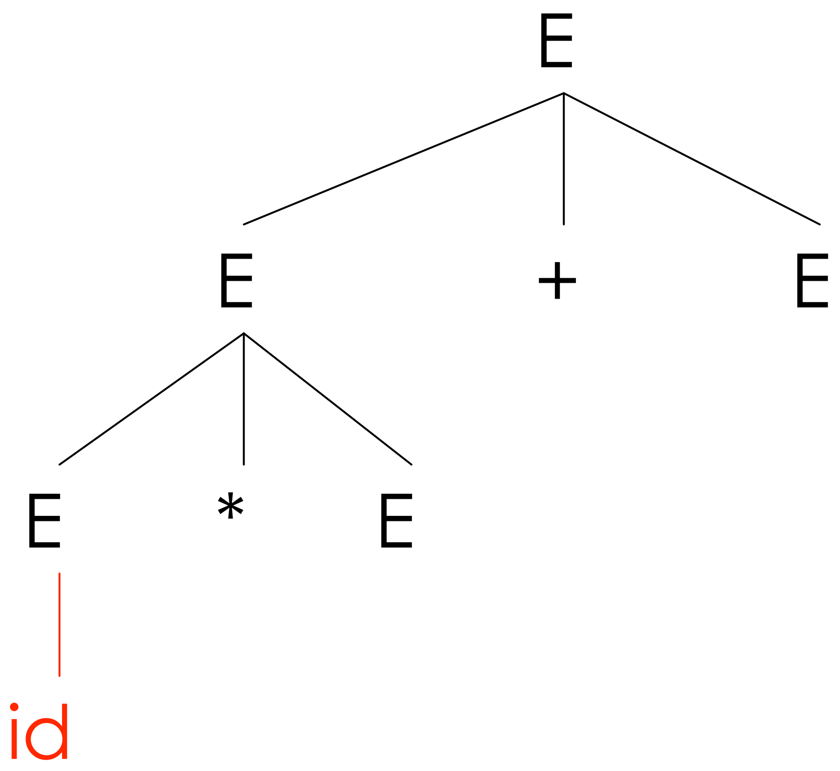
Derivation in Detail (3)

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



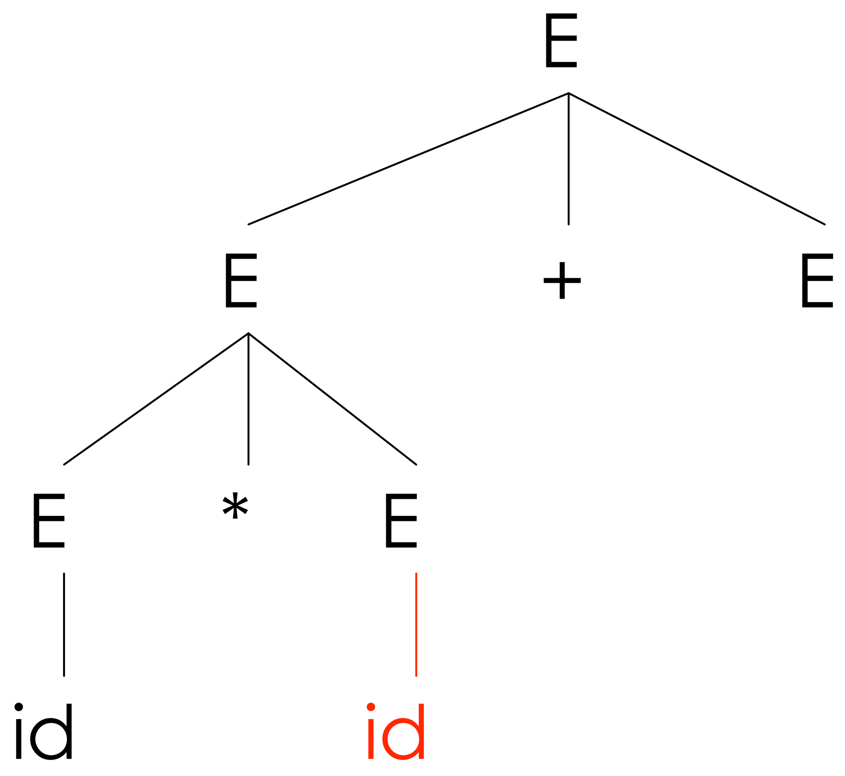
Derivation in Detail (4)

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



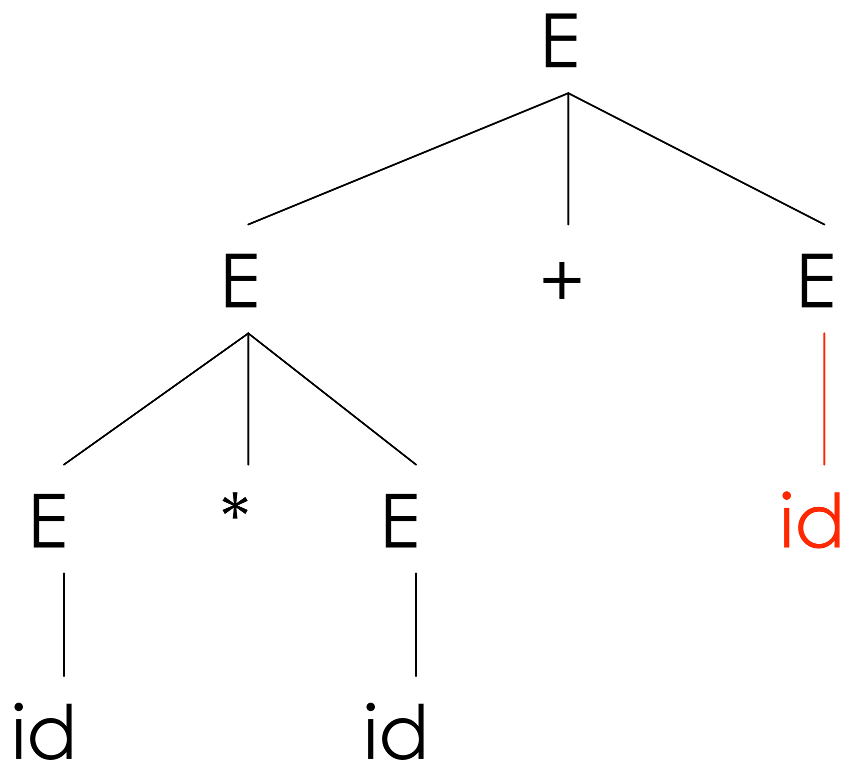
Derivation in Detail (5)

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



Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\quad & E+E \\
\rightarrow\quad & E * E+E \\
\rightarrow\quad & \mathrm{id} * E + E \\
\rightarrow\quad & \mathrm{id} * \mathrm{id} + E \\
\rightarrow\quad & \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}
$$



Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not

Left-most and Right-most Derivations

- The previous example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- Here is an equivalent notion of a *right-most* derivation ⇨

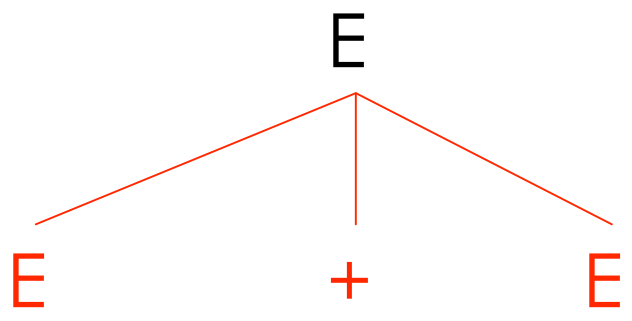
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+\mathrm{id} \\
\rightarrow\ & E * E + \mathrm{id} \\
\rightarrow\ & E * \mathrm{id} + \mathrm{id} \\
\rightarrow\ & \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}
$$

Right-most Derivation in Detail (1)

E

E

Right-most Derivation in Detail (2)

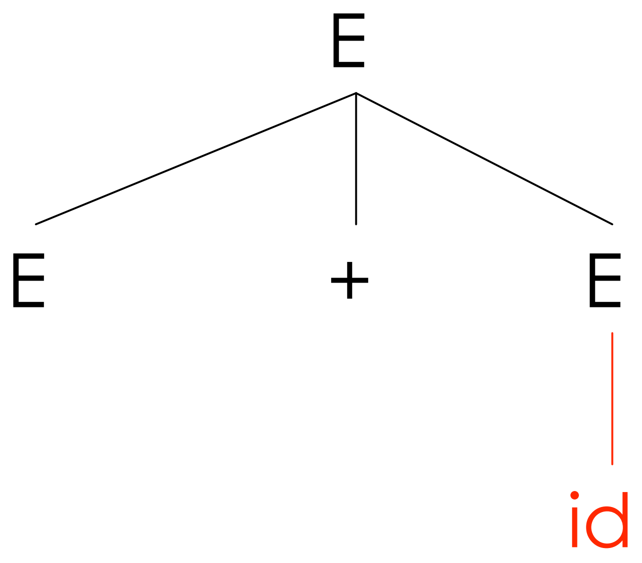


E

→ E+E

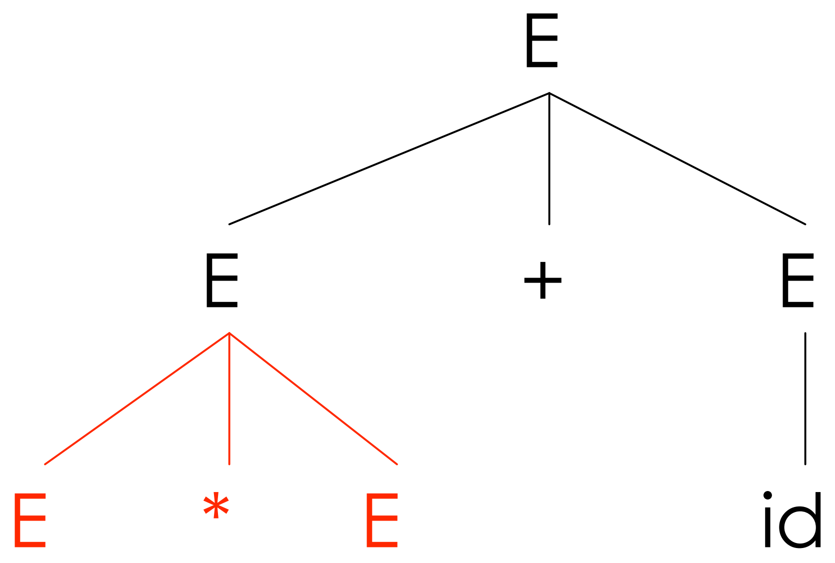
Right-most Derivation in Detail (3)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id
\end{aligned}
$$



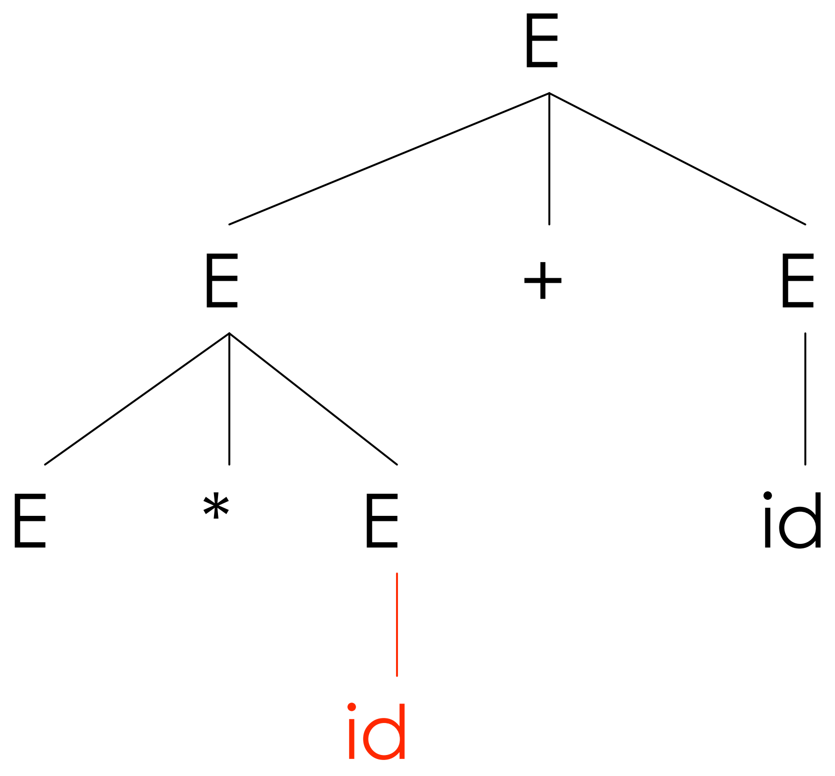
Right-most Derivation in Detail (4)

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



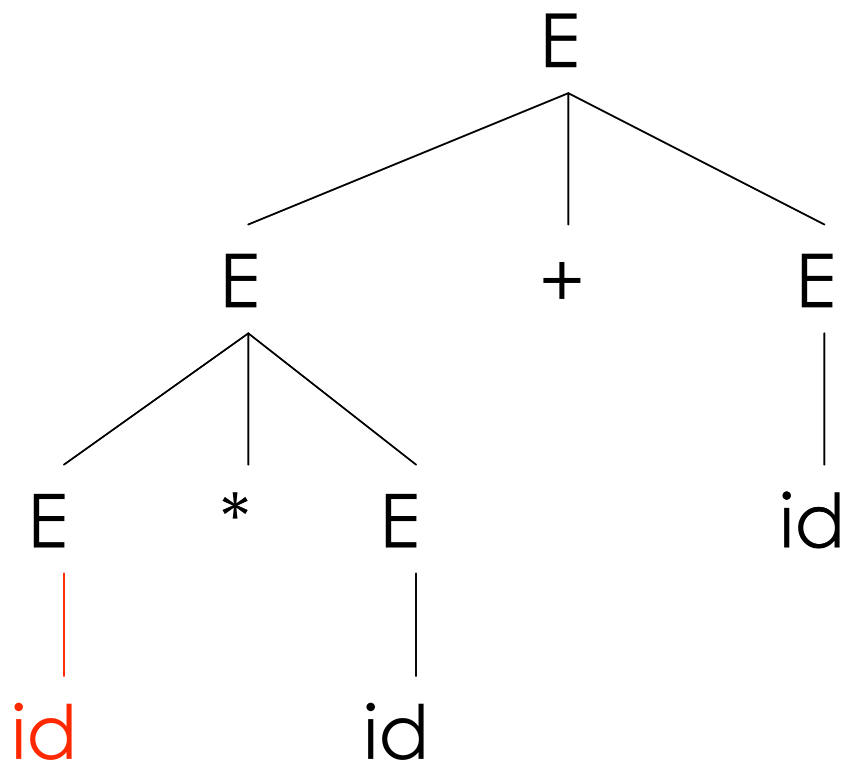
Right-most Derivation in Detail (5)

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



Right-most Derivation in Detail (6)

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



Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

Summary of Derivations

- We are not just interested in whether

$$s \in L(G)$$

  - We need a parse tree for s

- A derivation defines a parse tree
  - But one parse tree may have many derivations

- Left-most and right-most derivations are important in parser implementation