بسم الله الرحمن الرحيم

Parsing:

Context-Free Grammars, Parsing and Programming Languages

Languages and Automata

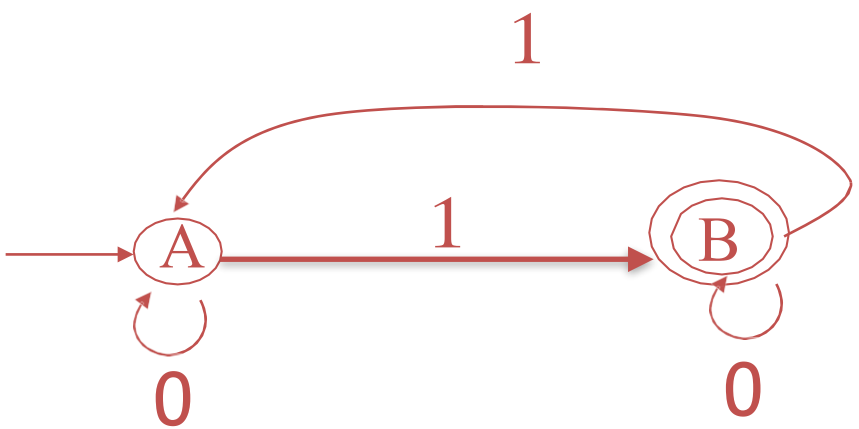
- Formal languages are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest formal languages widely used
  - Many applications
- We will also study context-free languages

Beyond Regular Languages

- Many languages are not regular
- Strings of balanced parentheses are not regular:

$$\left\{ (^i)^i \mid i \geq 0 \right\}$$

What Can Regular Languages Express?



What Can Regular Languages Express?

- Languages requiring counting modulo a fixed integer
- Intuition: *A* finite automaton that runs long enough must repeat states
- Finite automaton can't remember # of times it has visited a particular state

The Functionality of the Parser

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

  (But some parsers never produce a parse tree . . .)

Example

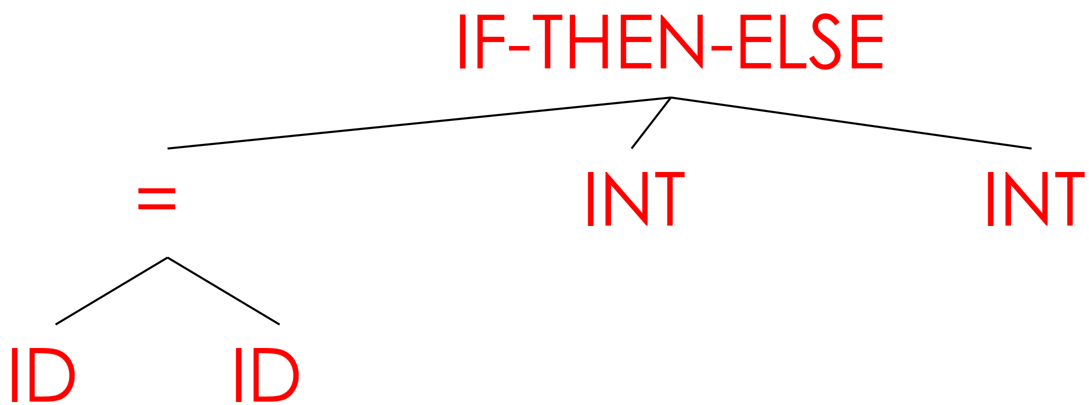
- Cool

  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	String of characters	String of tokens
Parser	String of tokens	Parse tree (may be implicit)

The Role of the Parser

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens

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

Context-Free Grammars

- Programming language constructs have recursive structure

- An STMT is

  if EXPR then STMT else STMT

  while EXPR do STMT end

  …

- 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

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$

$$\text{where } X \in N \text{ and } Y_i \in T \cup N \cup \{\varepsilon\}$$

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

Example of CFGs

Simple arithmetic expressions:

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

The Language of a CFG

Read productions as replacement rules:

$$X \rightarrow Y_1 \ldots Y_n$$

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

Key Idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal X in the string by a the 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

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

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 $L(G)$ of G is:

$$\{a_1 \ldots a_n \mid S \rightarrow^{*} a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal}\}$$

The sentential forms $SF(G)$ of G is:

$$\{X_1 \ldots X_n \mid S \rightarrow^{*} X_1 \ldots X_n \text{ and every } X_i \text{ is a terminal or non-terminal}\}$$

Therefore: $L(G) \subset SF(G)$

Terminals

- Terminals are so-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:

$$\begin{aligned} S &\rightarrow (S) \\ S &\rightarrow \varepsilon \end{aligned}$$

OR

$$\begin{aligned} S &\rightarrow (S) \\ &\mid \varepsilon \end{aligned}$$

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”; also need parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG’s (e.g., bison)

Derivations and Parse Trees

A derivation is a sequence of productions

$$S \to \ldots \to \ldots \to \ldots \to \ldots \to \ldots$$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \to Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

Derivation Example

- Grammar

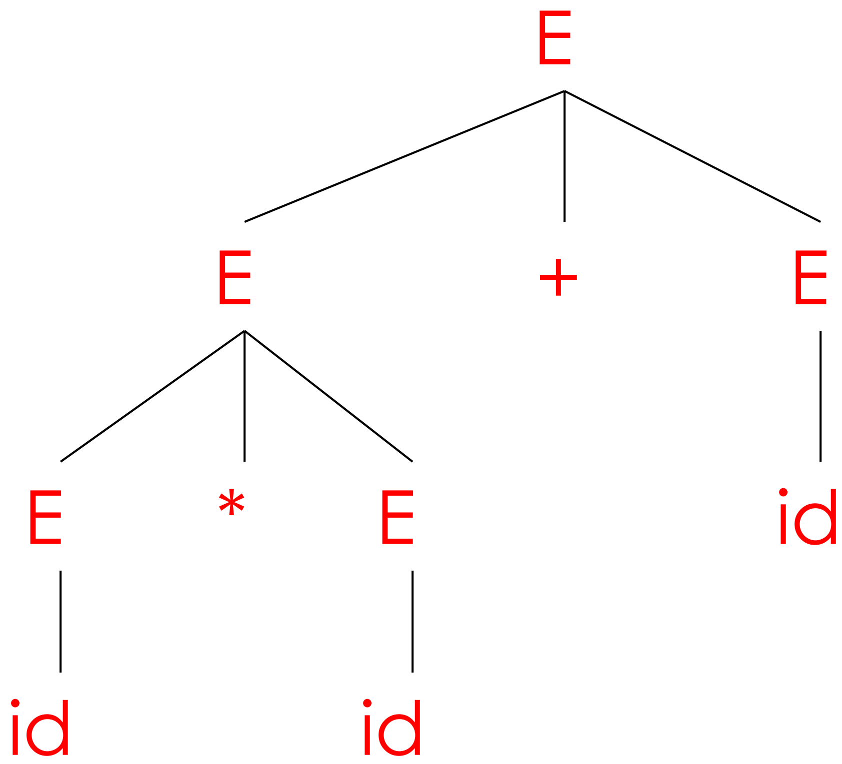
$$E \rightarrow E{+}E \mid E * E \mid (E) \mid \mathrm{id}$$

- String

$$\mathrm{id} * \mathrm{id} + \mathrm{id}$$

Derivation Example (Cont.)

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

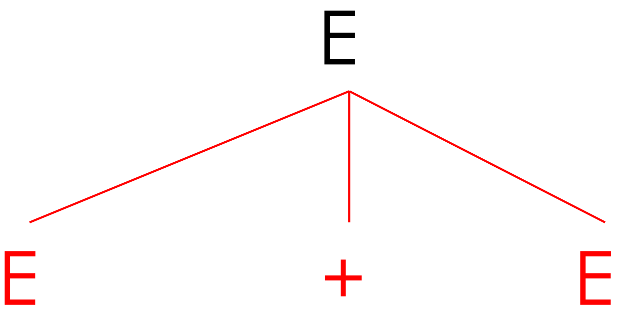


Derivation in Detail (1)

E

E

Derivation in Detail (2)



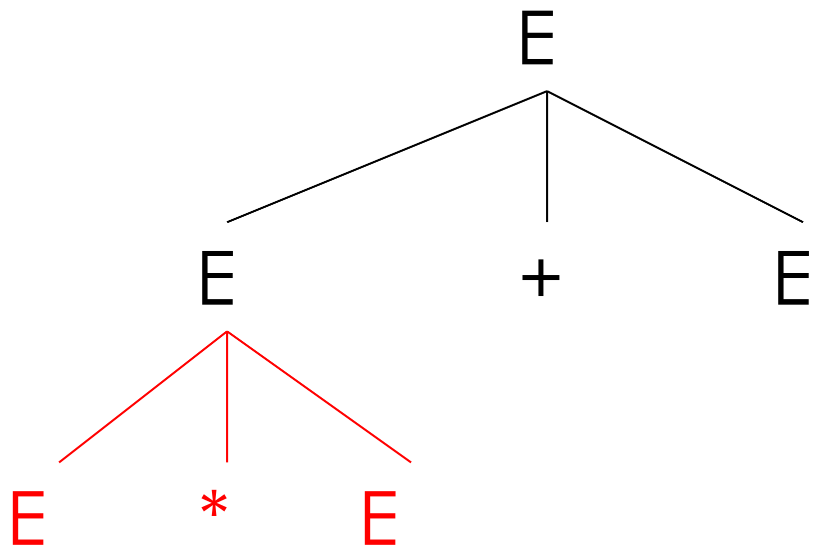
$$E$$
$$\rightarrow \quad E+E$$

Derivation in Detail (3)

$$E$$

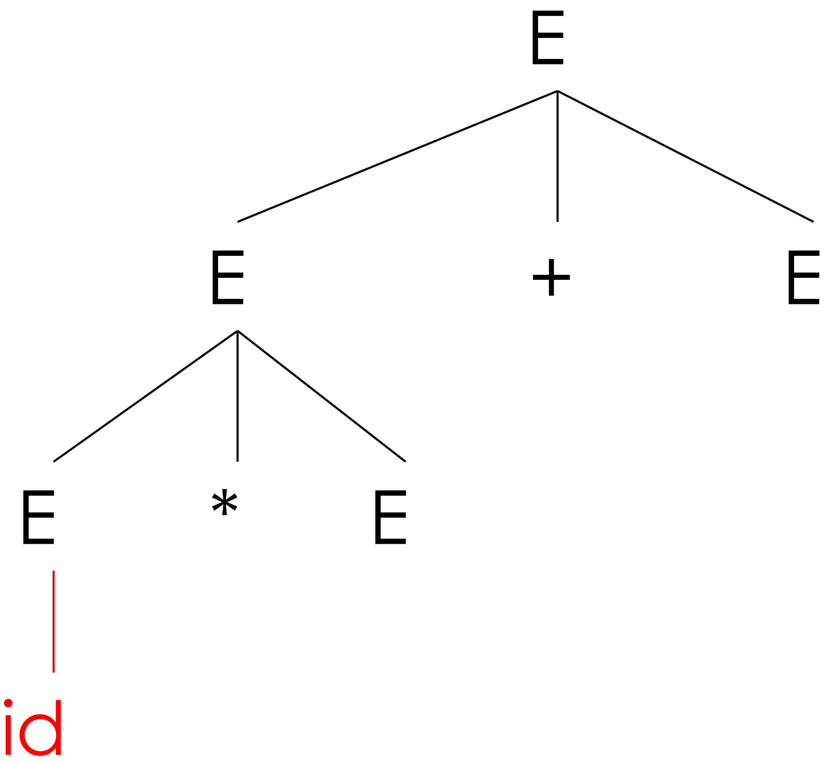
$$\rightarrow \quad E+E$$

$$\rightarrow \quad E * E+E$$



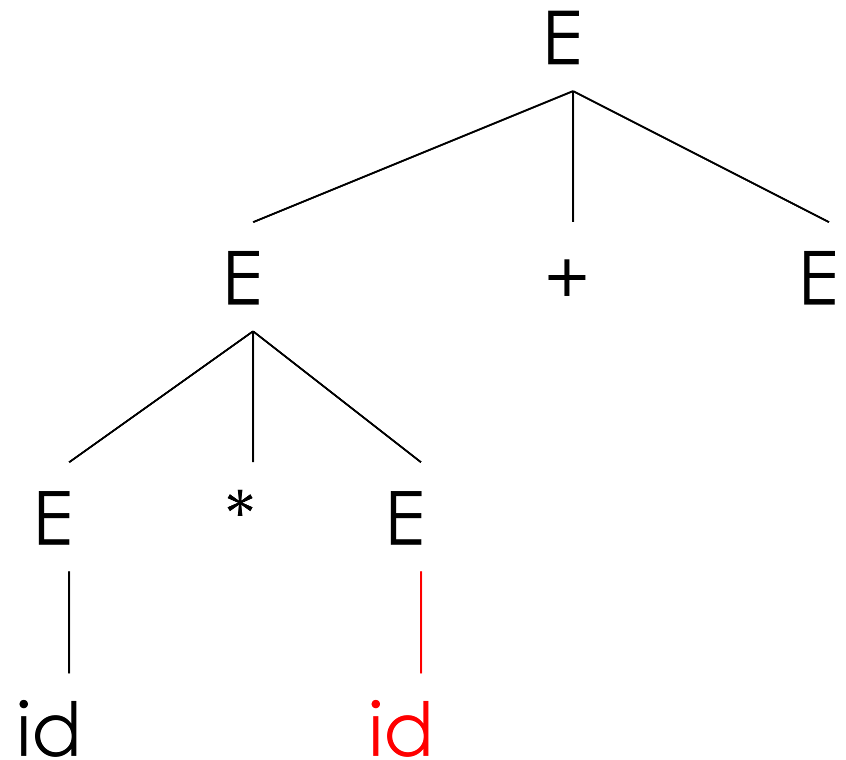
Derivation in Detail (4)

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



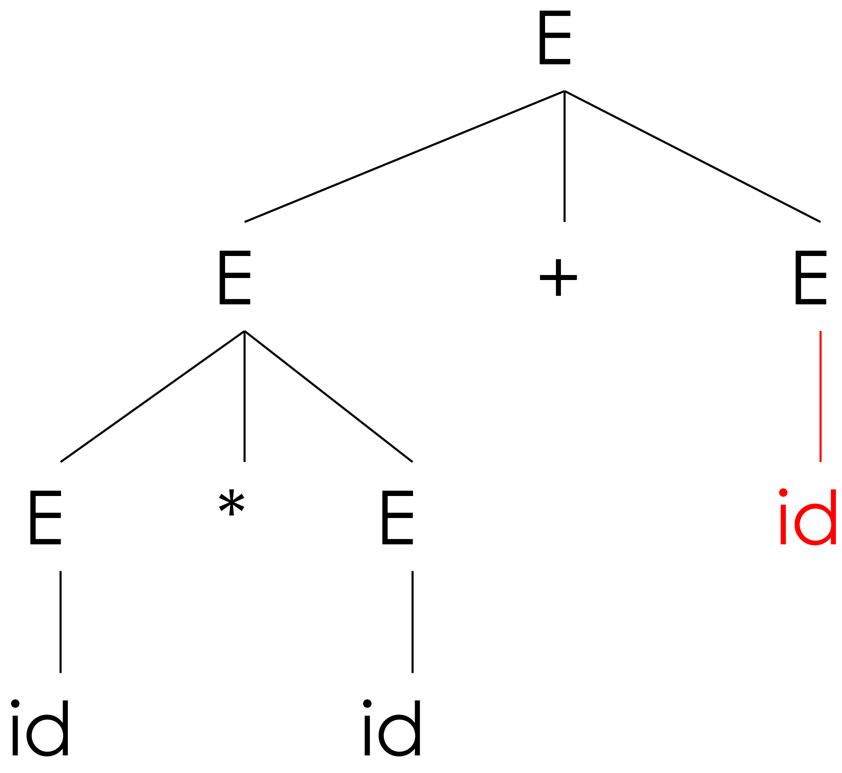
Derivation in Detail (5)

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



Derivation in Detail (6)

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



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 example was a left-most derivation
  - At each step, replaced the left-most non-terminal
- There is an equivalent notion of a right-most derivation

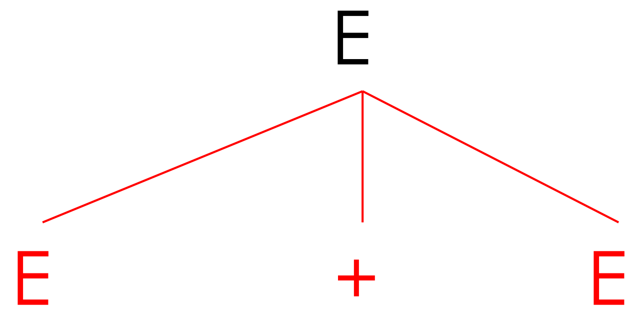
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$

Right-most Derivation in Detail (1)

E

E

Right-most Derivation in Detail (2)

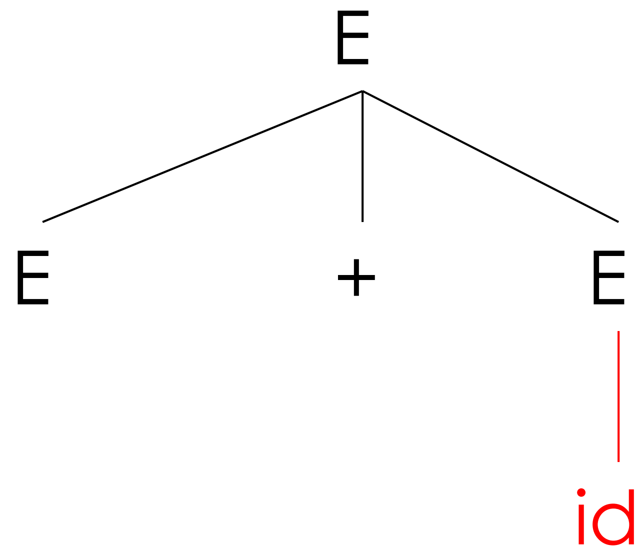


$$E$$

$$\rightarrow \quad E+E$$

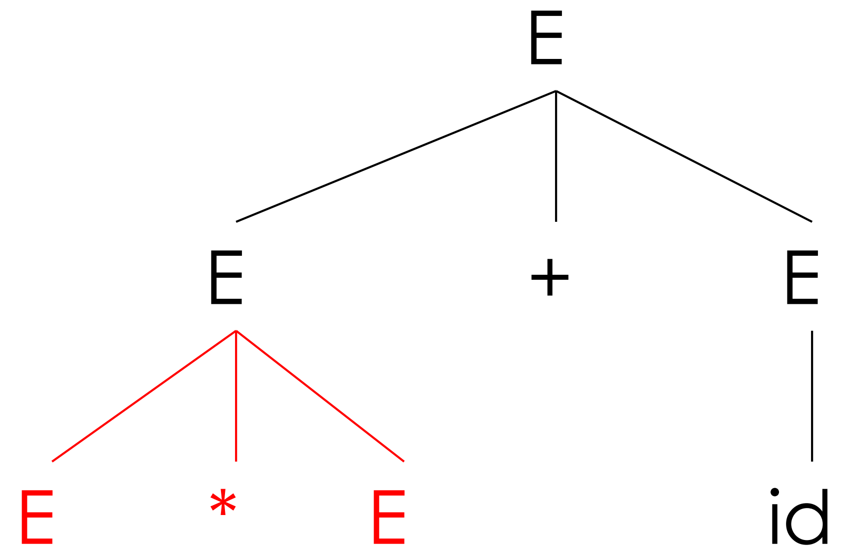
Right-most Derivation in Detail (3)

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



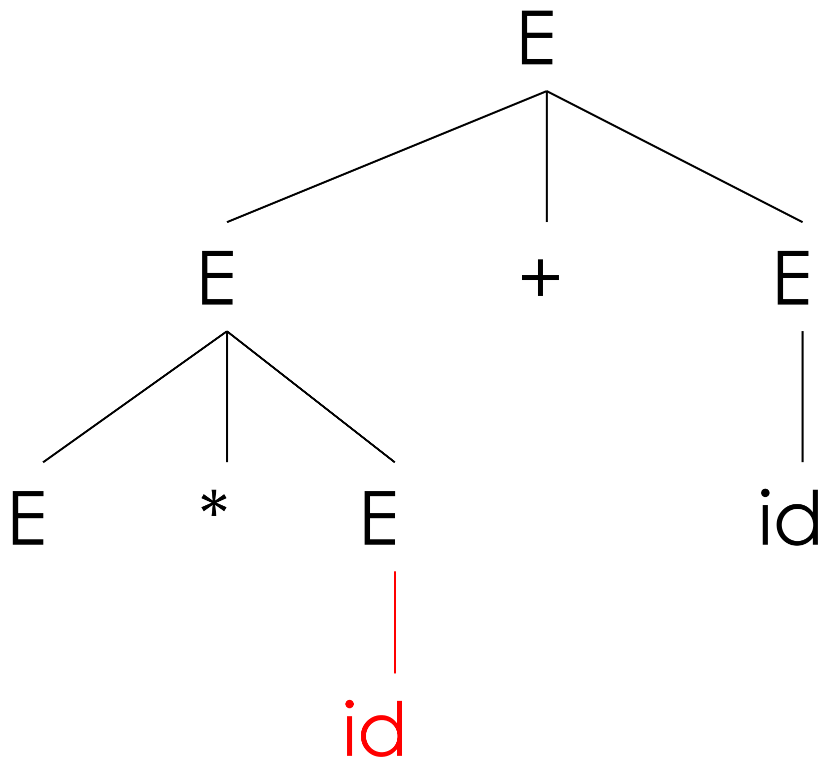
Right-most Derivation in Detail (4)

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



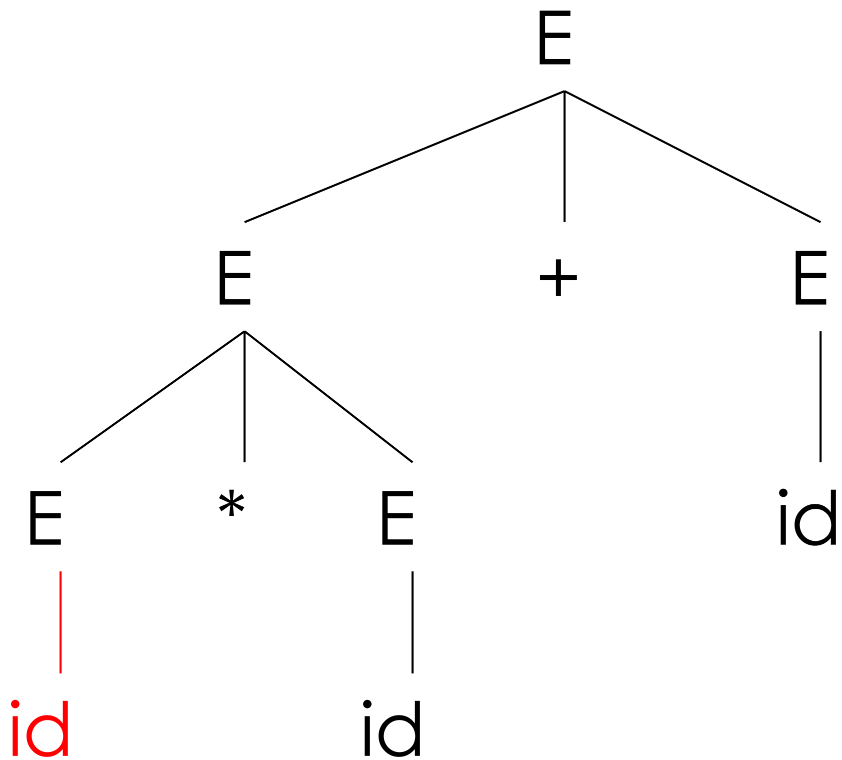
Right-most Derivation in Detail (5)

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



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

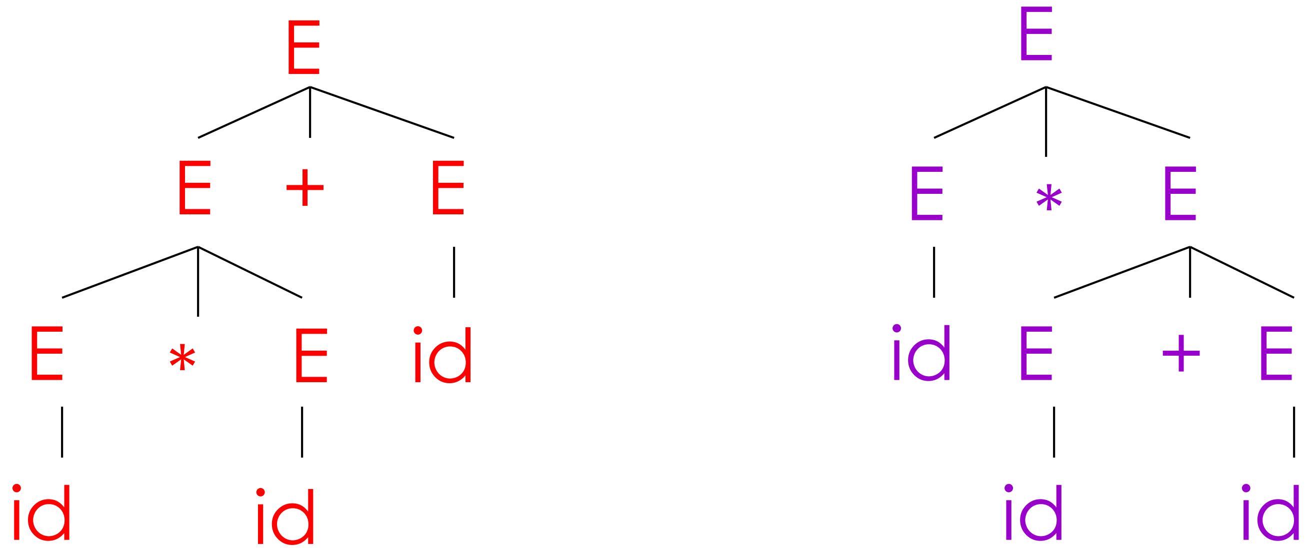
- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Ambiguity

- Grammar $E \rightarrow E + E \mid E * E \mid (E) \mid id$
- String $id * id + id$

Ambiguity (Cont.)

This string has two parse trees



Ambiguity (Cont.)

- A grammar is ambiguous if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string

- Ambiguity is **BAD**
  - Leaves meaning of some programs ill-defined

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite grammar unambiguously

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

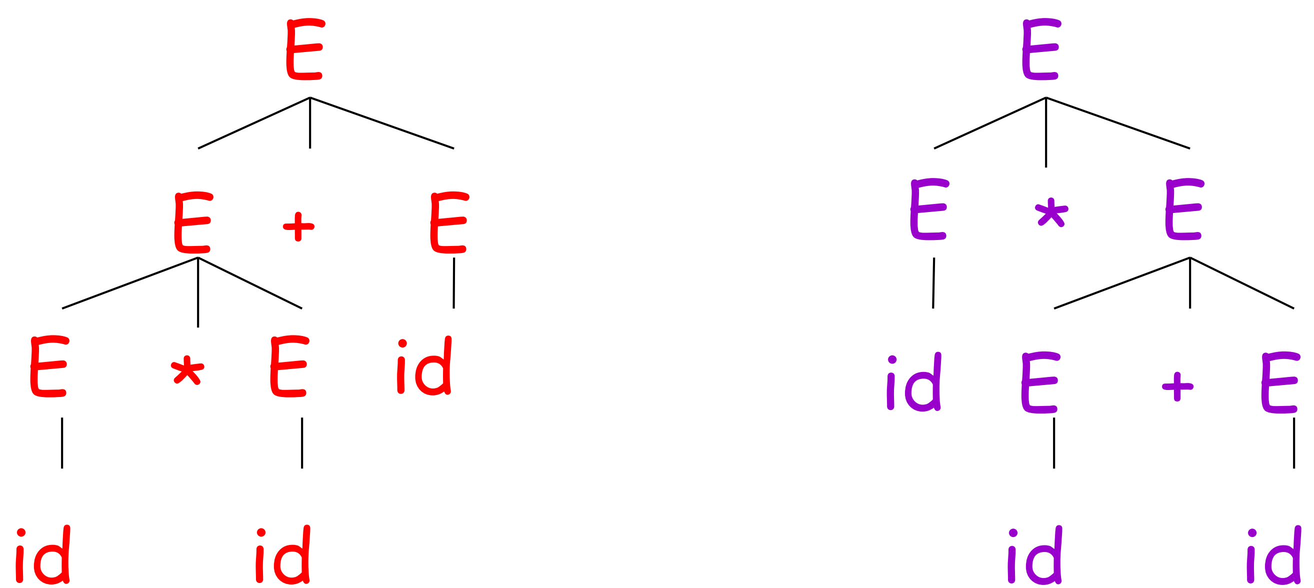
- Enforces precedence of * over +

Ambiguity in Arithmetic Expressions

- Recall the grammar

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

- The string id * id + id has two parse trees:

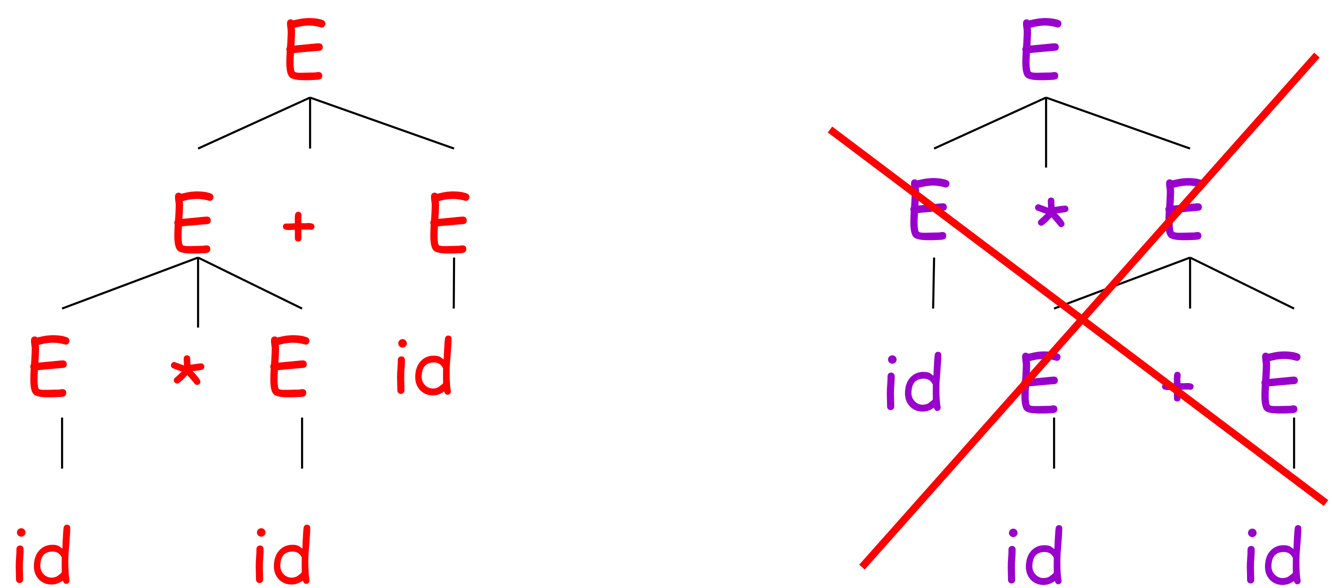


Ambiguity in Arithmetic Expressions

- Recall the grammar

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

- The string $id * id + id$ has two parse trees:



Ambiguity: The Dangling Else

- Consider the grammar

$$
\begin{aligned}
S \rightarrow\ & \text{if } E \text{ then } S \\
\mid\ & \text{if } E \text{ then } S \text{ else } S \\
\mid\ & \text{OTHER}
\end{aligned}
$$

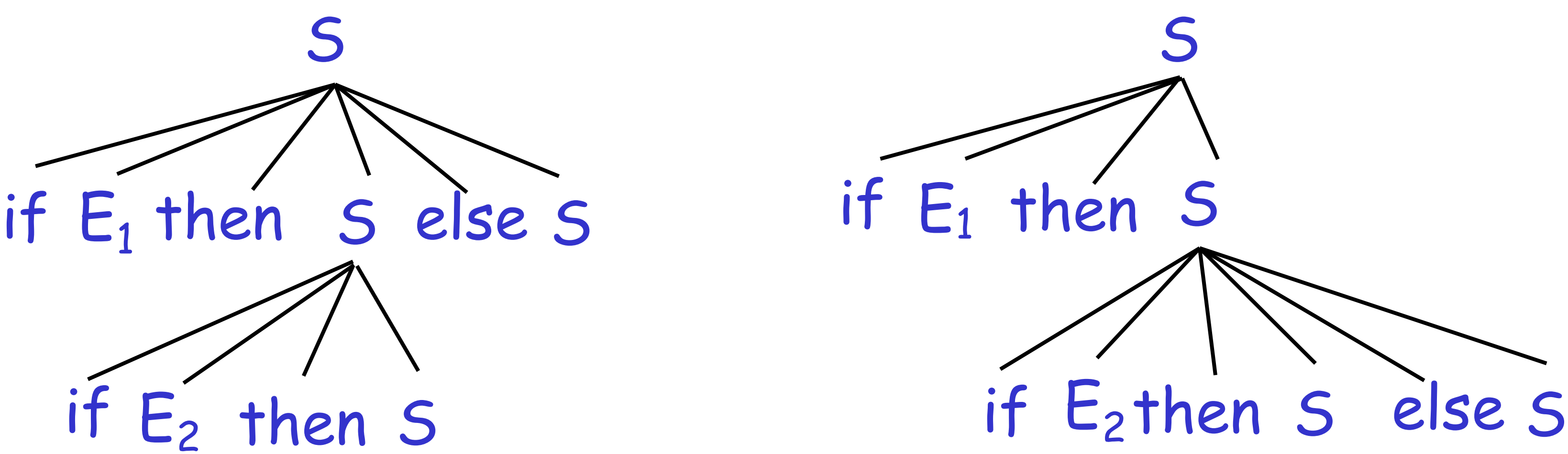
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then S else S

  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

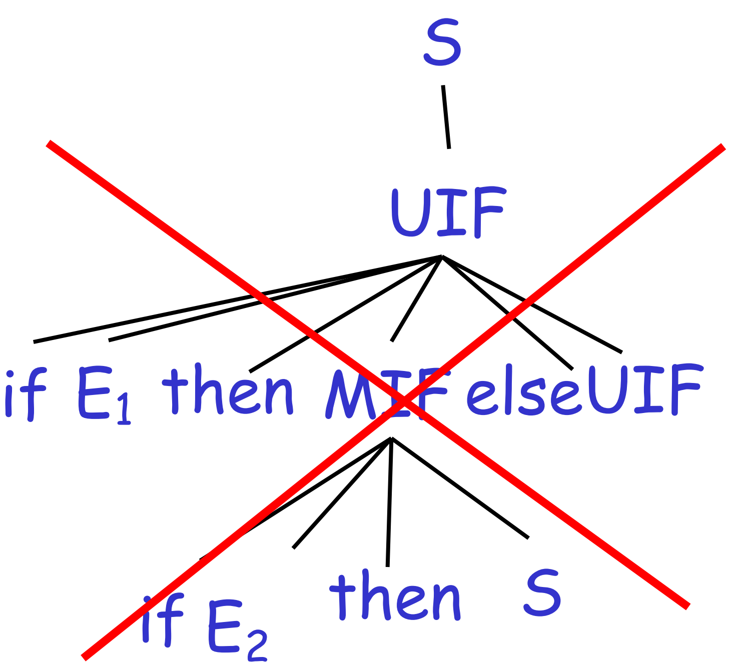
- else matches the closest unmatched then
- We can describe this in the grammar

$S \to$ MIF /* all then are matched */
    | UIF /* some then is unmatched */
MIF $\to$ if E then MIF else MIF
    | OTHER
UIF $\to$ if E then S
    | if E then MIF else UIF

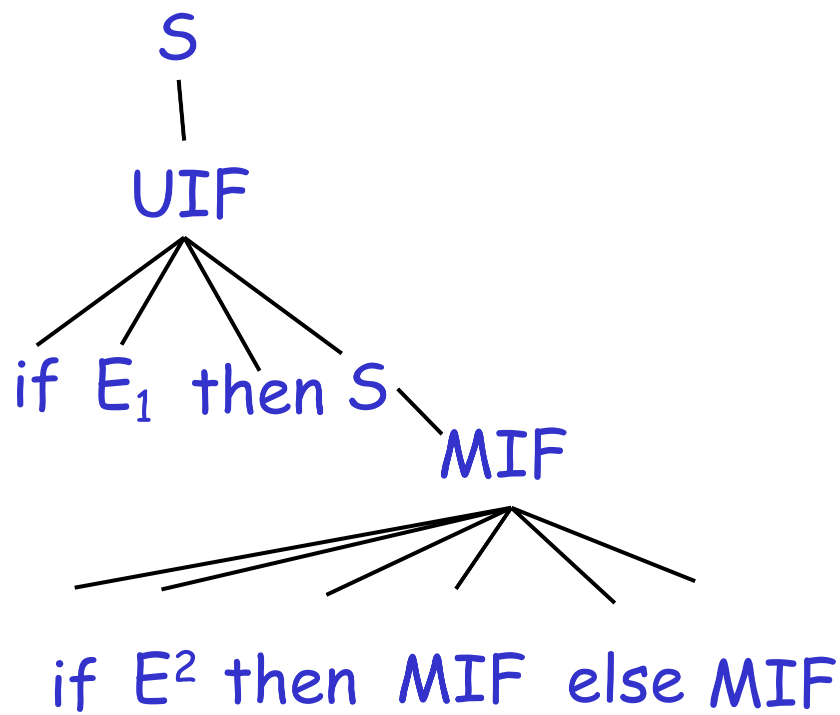
- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then S else S



- Not valid because the then expression is not a MIF



- A valid parse tree (for a UIF)