Introduction to Parsing

CS143

Lecture 5

Instructor: Fredrik Kjolstad

Slide design by Prof. Alex Aiken, with modifications

Outline

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

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 today 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?

- 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
  (Conceptually, but in practice parsers return an AST)

Example

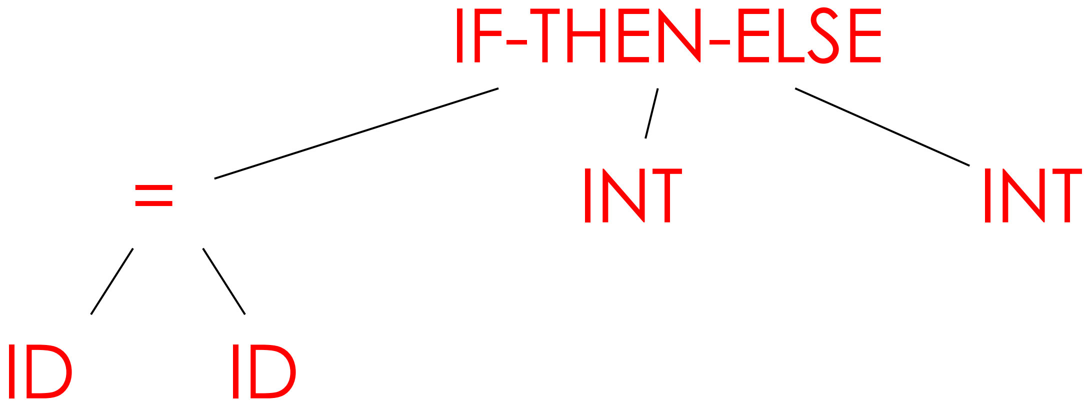
- Cool

if x = y then 1 else 2 fi

- Parser input

IF ID = ID THEN INT ELSE INT FI

- Parser output



IF-THEN-ELSE

=

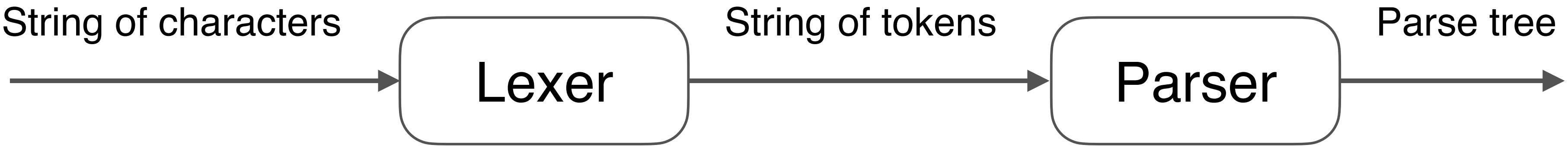
INT

INT

ID

ID

Comparison with Lexical Analysis



String of characters

Lexer

String of tokens

Parser

Parse tree

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 EXPR is

  if EXPR then EXPR else EXPR fi

  while EXPR loop EXPR pool

  …

- 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

$$X \to Y_1 Y_2 \ldots Y_n$$

where $X \in N$ 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

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:

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

(Running example this week and next)

The Language of a CFG

Read productions as 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 of G is:

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

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:

$$\begin{array}{ccc} S & \rightarrow & (S) \\ S & \rightarrow & \varepsilon \end{array} \quad \text{OR} \quad \begin{array}{ccc} S & \rightarrow & (S) \\ & \mid & \varepsilon \end{array}$$

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 Cool CFG

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id fi

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)

$(\mathrm{id}) * \mathrm{id}$

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

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

$\mathrm{id} * (\mathrm{id})$

Notes

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

- Membership in a language is “yes” or “no”
  - We also need a 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 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 leading to a string of only terminals

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

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

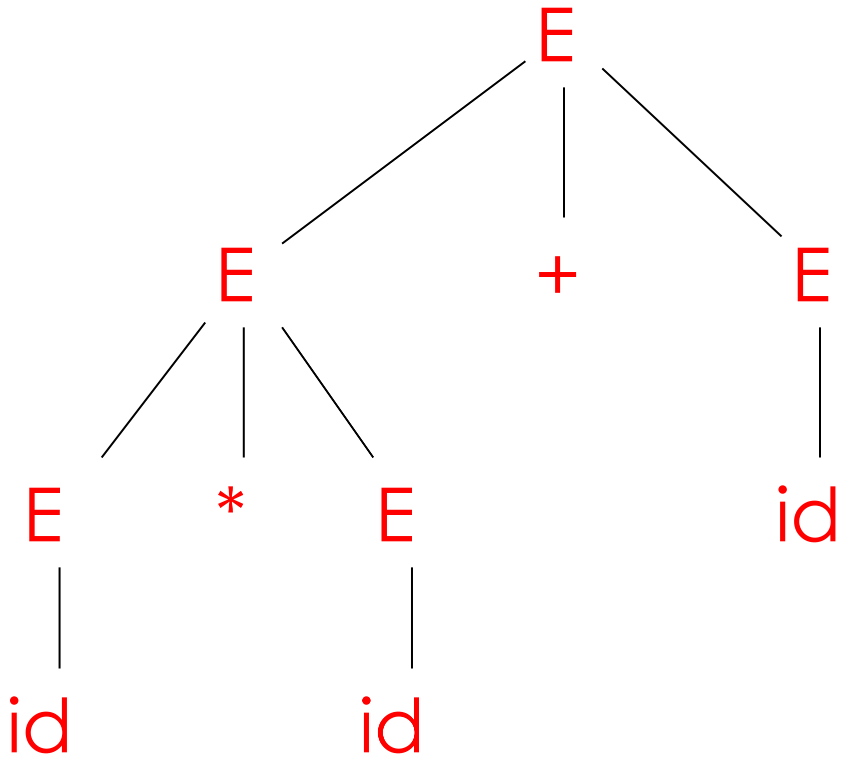
  $$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

- String

  $$id * id + id$$

Derivation Example (cont.)

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

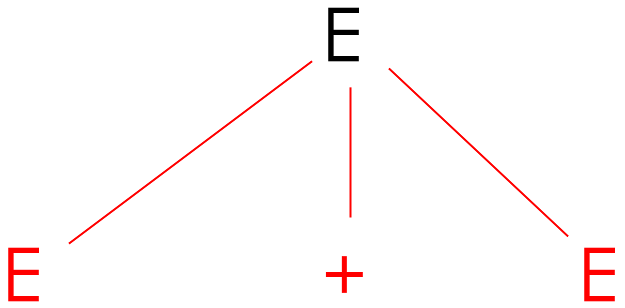


Derivation in Detail (1)

E

E

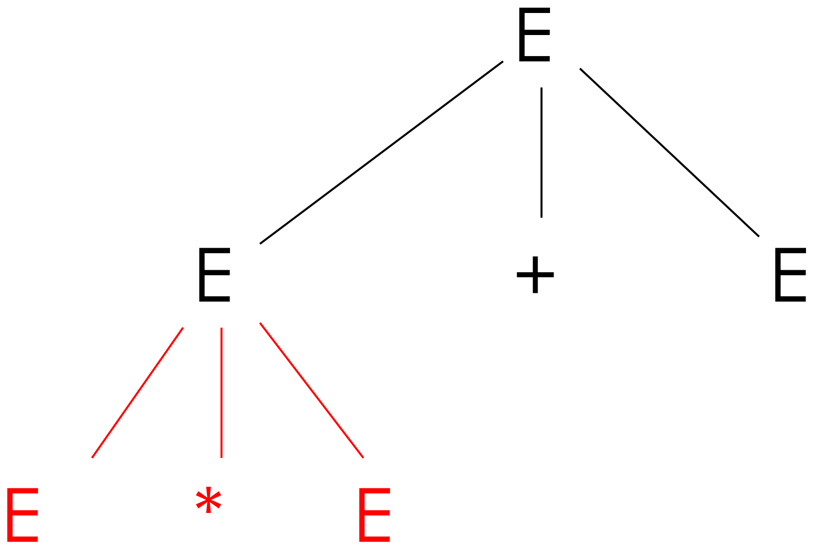
Derivation in Detail (2)



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

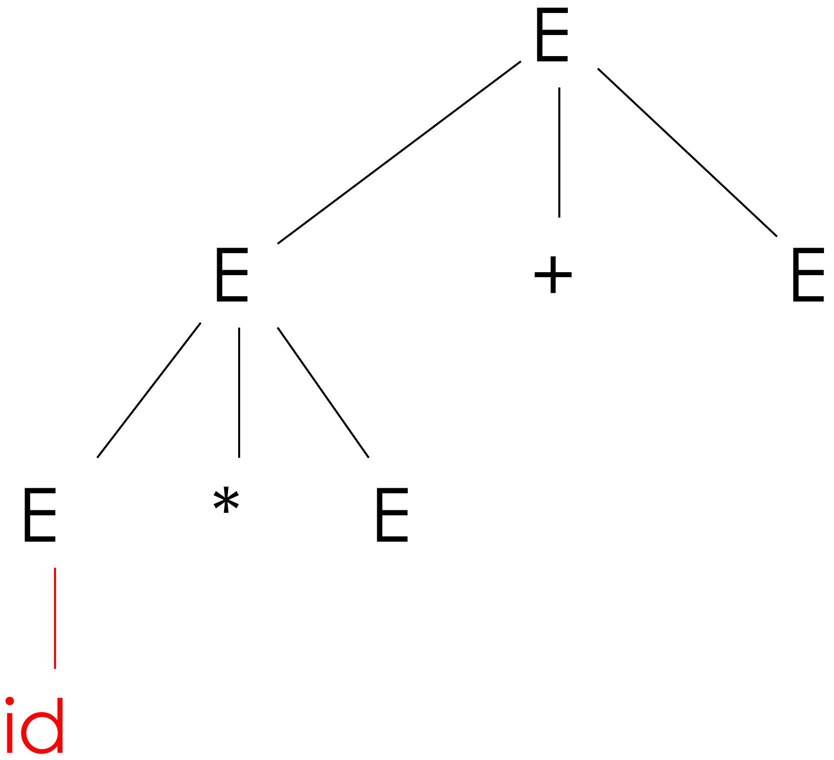
Derivation in Detail (3)

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



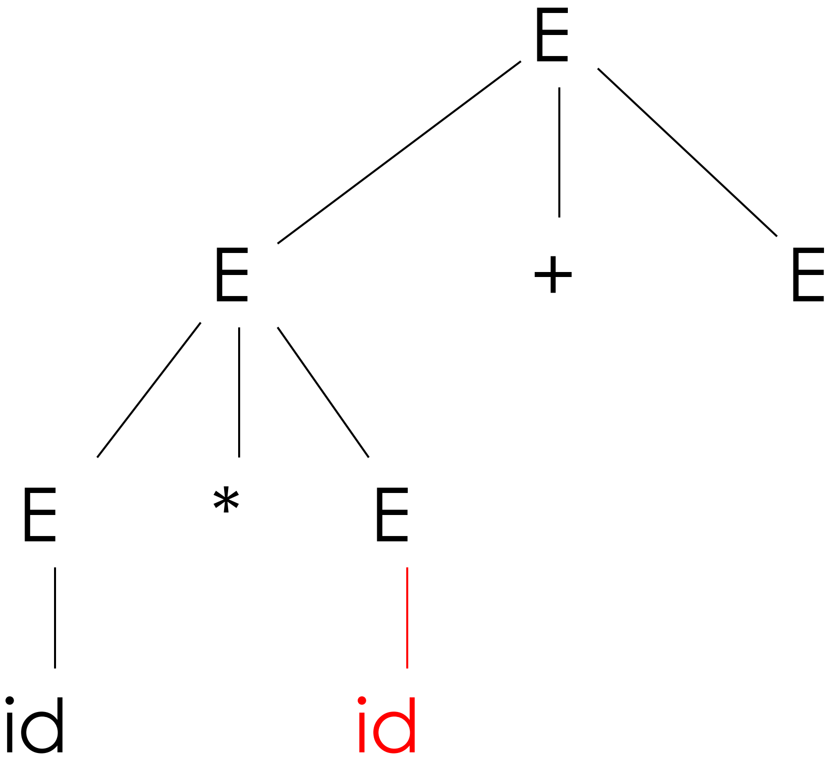
Derivation in Detail (4)

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



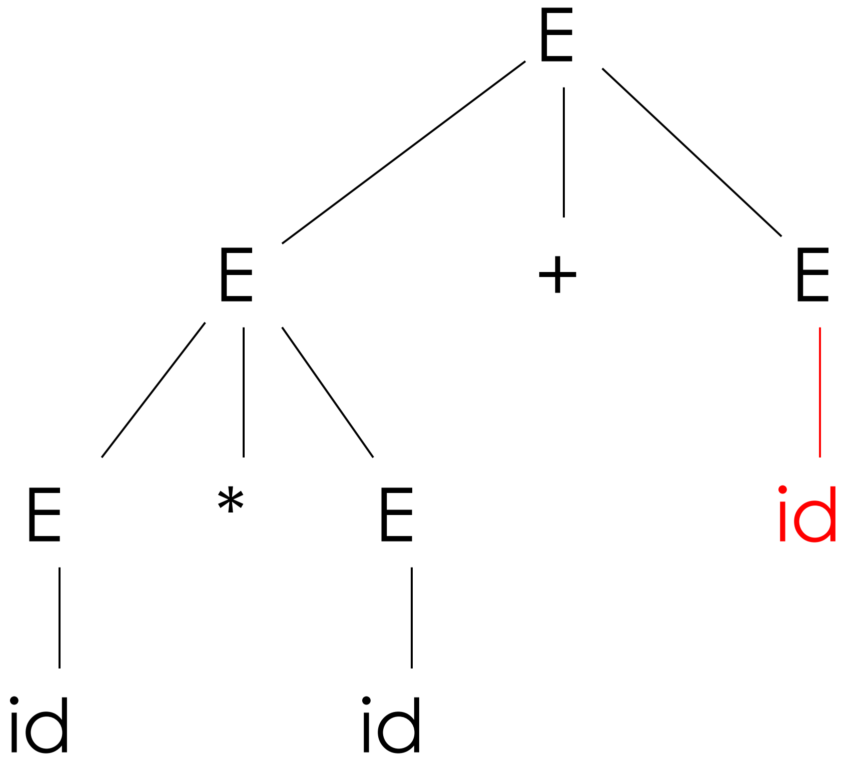
Derivation in Detail (5)

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



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

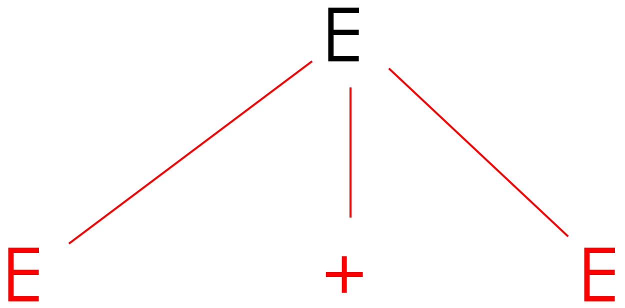
$$\begin{aligned}
& \mathrm{E} \\
\rightarrow\ & \mathrm{E+E} \\
\rightarrow\ & \mathrm{E+id} \\
\rightarrow\ & \mathrm{E} * \mathrm{E} + \mathrm{id} \\
\rightarrow\ & \mathrm{E} * \mathrm{id} + \mathrm{id} \\
\rightarrow\ & \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}$$

Right-most Derivation in Detail (1)

E

E

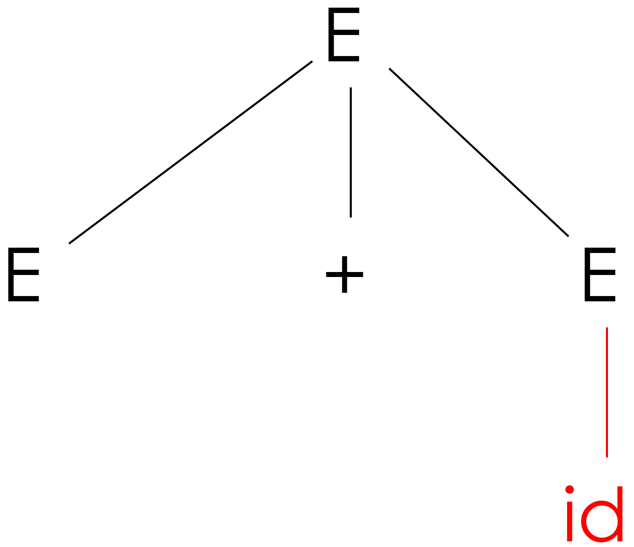
Right-most Derivation in Detail (2)



$$
\begin{array}{ll}
 & E \\
\rightarrow & E+E
\end{array}
$$

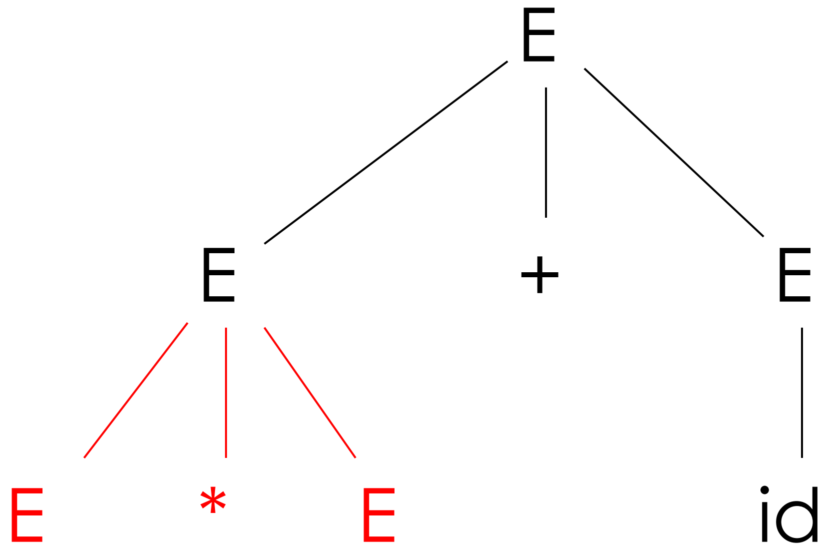
Right-most Derivation in Detail (3)

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



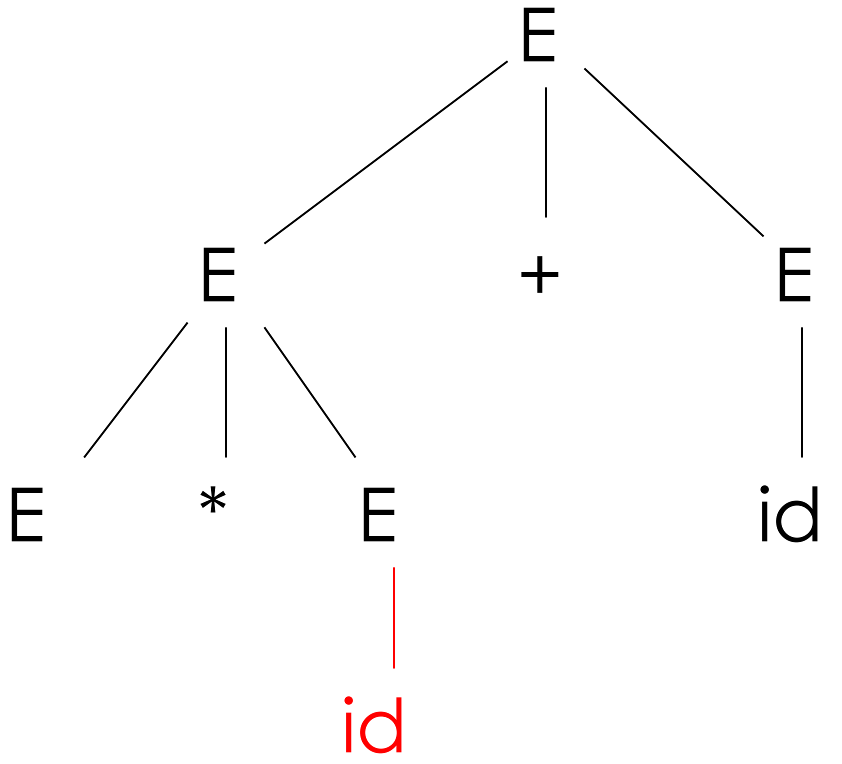
Right-most Derivation in Detail (4)

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



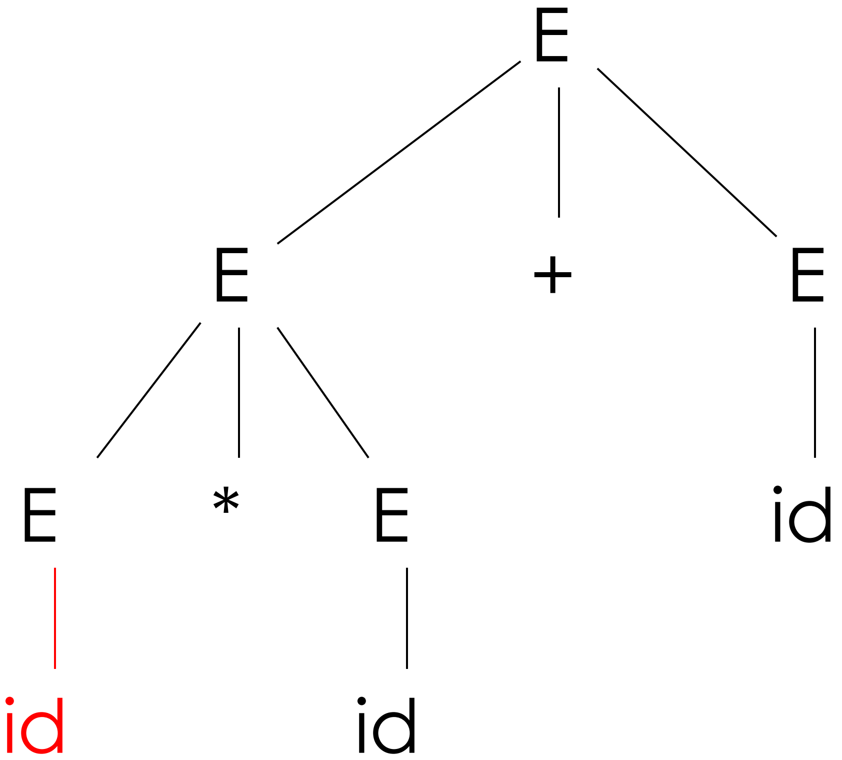
Right-most Derivation in Detail (5)

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



Right-most Derivation in Detail (6)

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



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

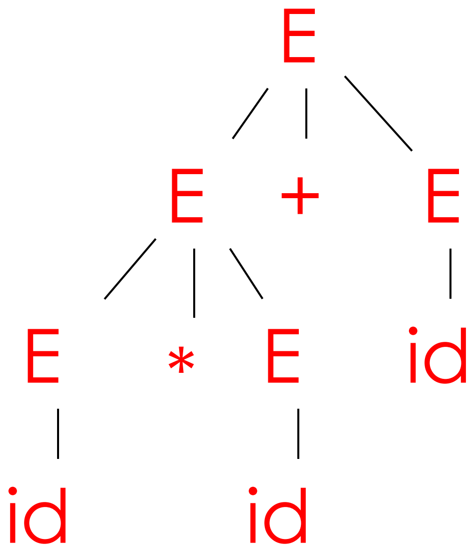
- Left-most and right-most derivations are important in parser implementation

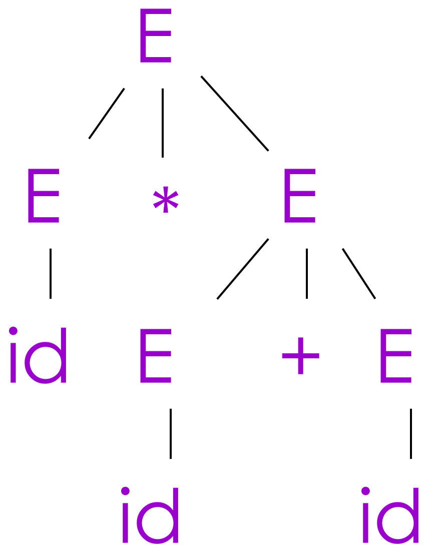
Ambiguity

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

- There are several ways to handle ambiguity

- Most direct method is to rewrite grammar unambiguously

$$\begin{aligned} E &\rightarrow E' + E \mid E' \\ E' &\rightarrow \mathrm{id} * E' \mid \mathrm{id} \mid (E) * E' \mid (E) \end{aligned}$$

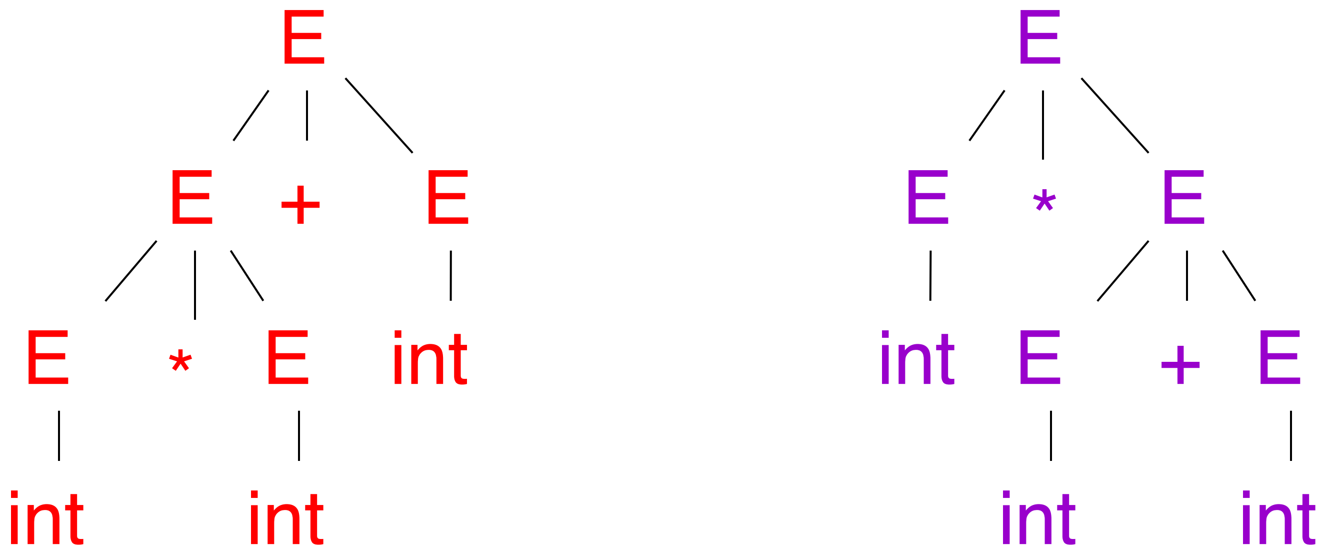
- Enforces precedence of * over +

Ambiguity in Arithmetic Expressions

- Recall the grammar

  E → E + E | E * E | (E) | int

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



Ambiguity: The Dangling Else

- Consider the grammar

```
E → if E then E
    | if E then E else E
    | OTHER
```

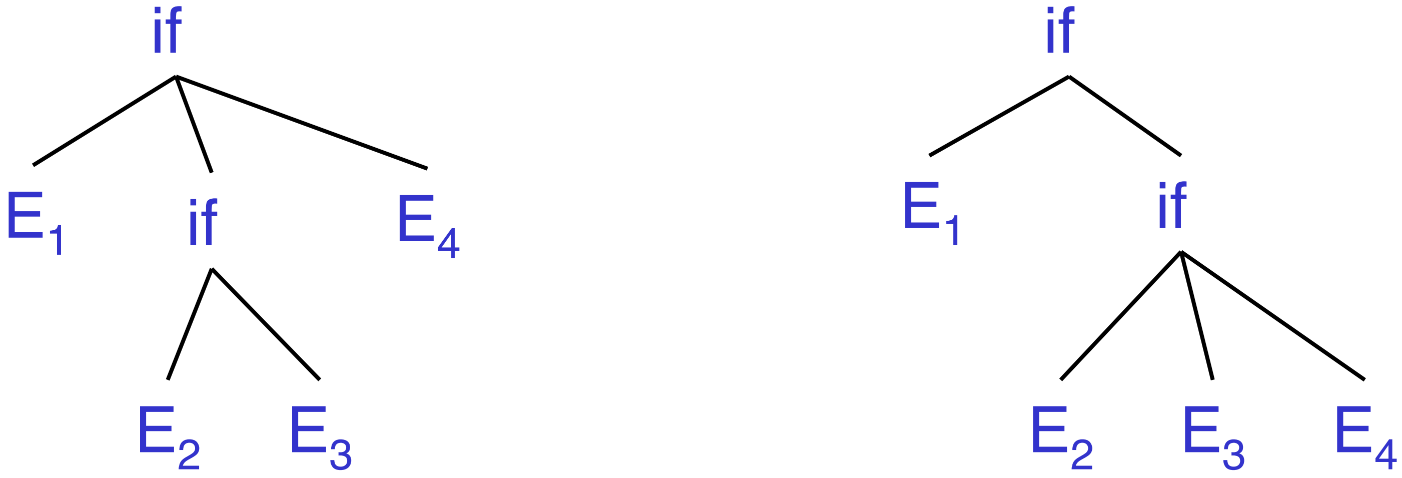
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees



- Typically we want the second form

E → if E then E
    | if E then E else E
    | OTHER

The Dangling Else: A Fix

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

```
E → MIF          /* all then are matched */
  | UIF          /* some then is unmatched */
```

```
MIF → if E then MIF else MIF
    | OTHER
```

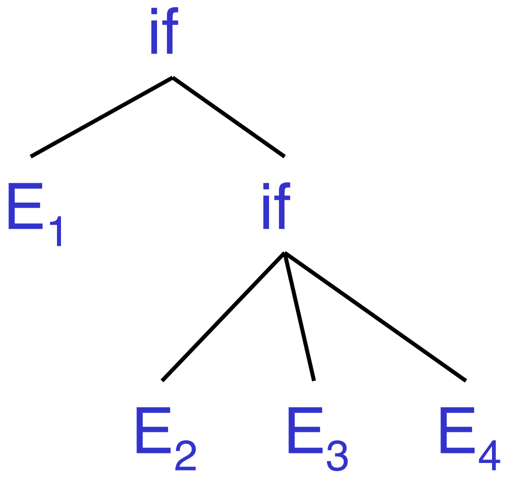
Key: Disallow if-then inside then-clause

```
UIF → if E then E
    | if E then MIF else UIF
```

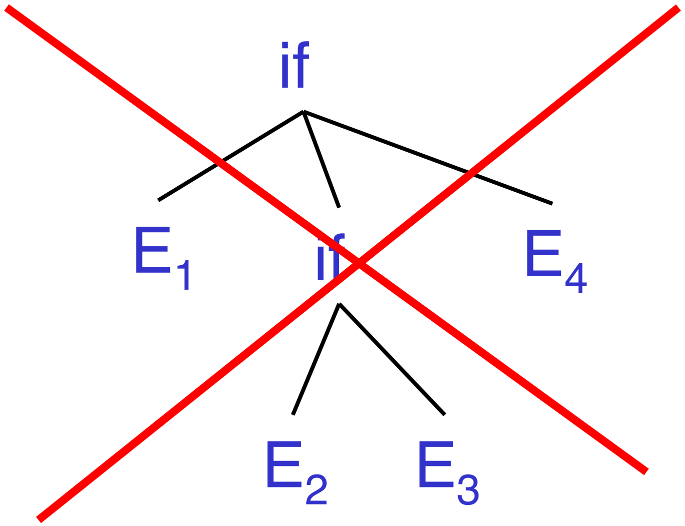
The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then E_3 else E_4

UIF → if E then E
| if E then MIF else UIF



- A valid parse tree (for a UIF)



- Not valid because the then expression is not a MIF

Ambiguity

- No general techniques for handling ambiguity

- Impossible to convert automatically an ambiguous grammar to an unambiguous one

- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

Precedence and Associativity Declarations

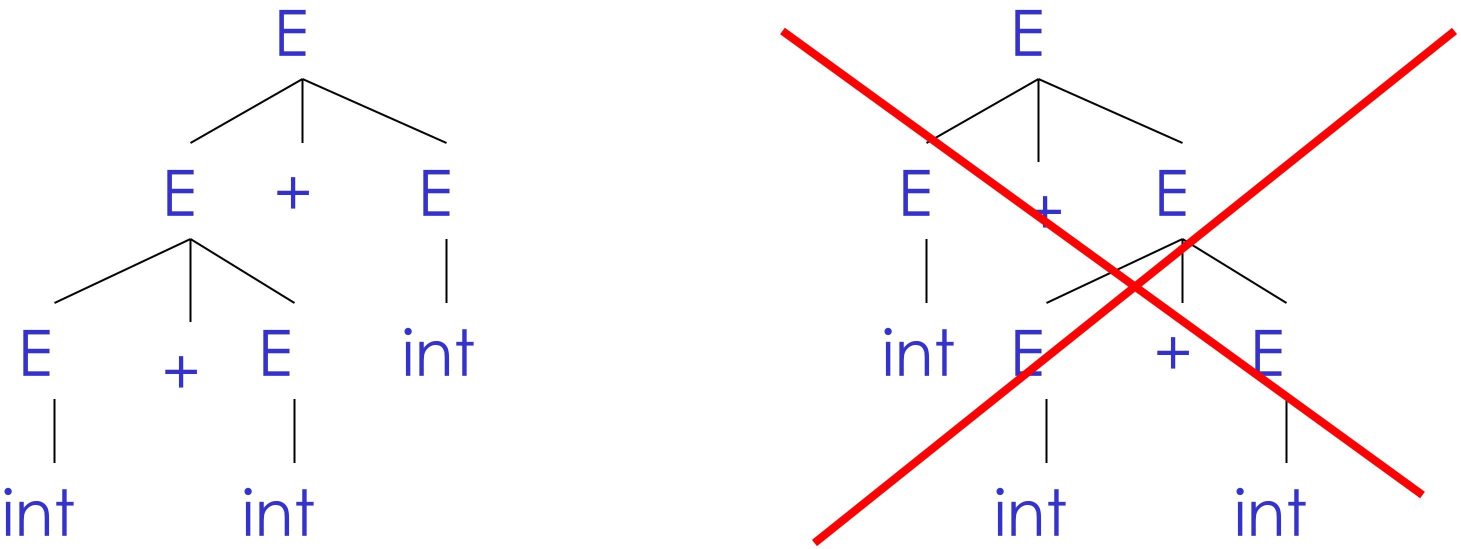
- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations

- Most tools allow precedence and associativity declarations to disambiguate grammars

- Examples …

Associativity Declarations

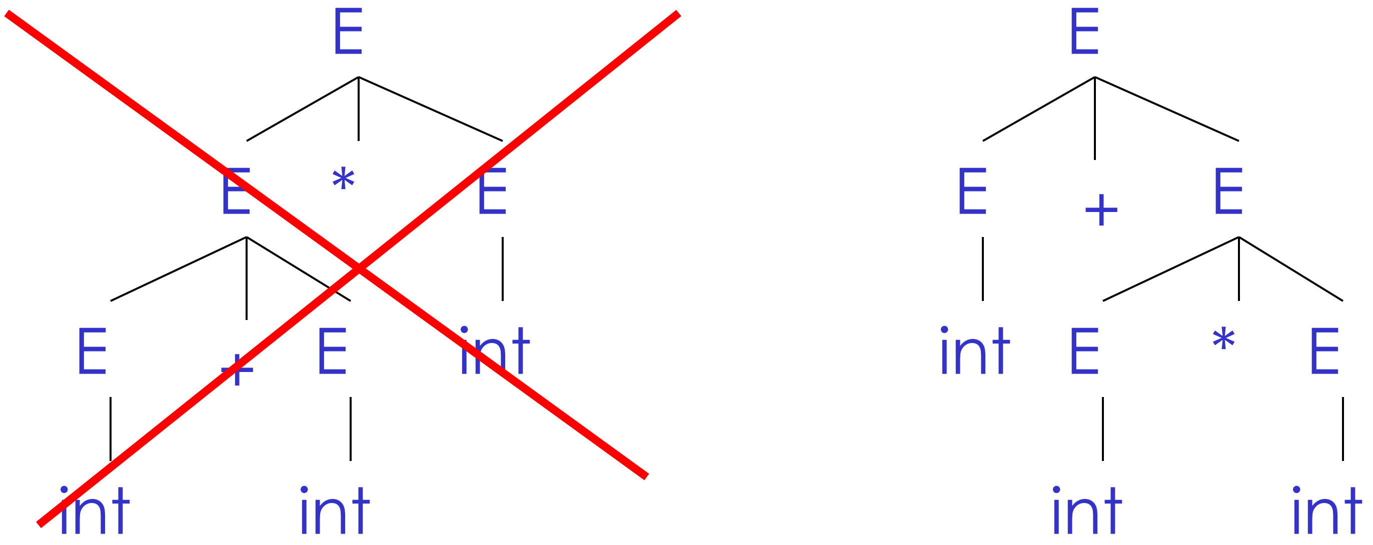
- Consider the grammar E → E + E | int
- Ambiguous: two parse trees of int + int + int



- Left associativity declaration: %left +

Precedence Declarations

- Consider the grammar E → E + E | E * E | int
  - And the string int + int * int



- Precedence declarations: %left +

  %left *