Lecture 4
Introduction to Parsing

Outline

- Regular languages revisited
- 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 also study context-free languages, tree languages

Beyond Regular Languages

- Many languages are not regular
- Strings of balanced parentheses are not regular:
  $\{ (^i\ )^i \mid i \geq 0\}$

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
- Finite automaton has finite memory
  - Only enough to store in which state it is
  - Cannot count, except up to a finite limit

The Functionality of the Parser

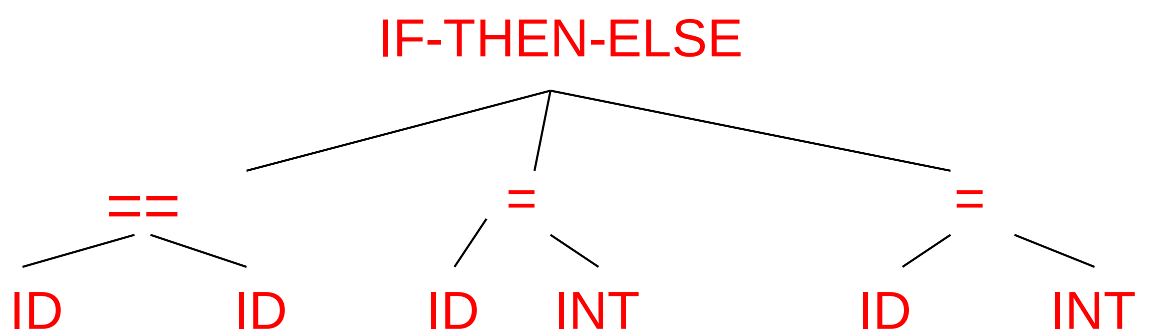
- **Input:** sequence of tokens from lexer

- **Output:** parse tree of the program

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

Example

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



Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	string of characters	string of tokens
Parser	string of tokens	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 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
- Context-free grammars are a natural notation for this recursive structure

Context-Free Grammars

- 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 $Y_i \in N \cup T \cup \{\varepsilon\}$

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

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.

Examples of CFGs

- Simple arithmetic expressions:

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

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 right-hand side of some production
$$X \to Y_1 \dots Y_n$$
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:

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

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 $\{(^i)^i \mid i>=0\}$

- Two grammars:
- $S \rightarrow (S)$
- $S \rightarrow \varepsilon$

- OR $S \rightarrow (S) \mid \varepsilon$

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!)

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow \text{int}$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$

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 procutions

  $S \to \ldots \to \ldots$

- 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

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

- String

$$int * int + int$$

Derivation Example (Cont.)

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

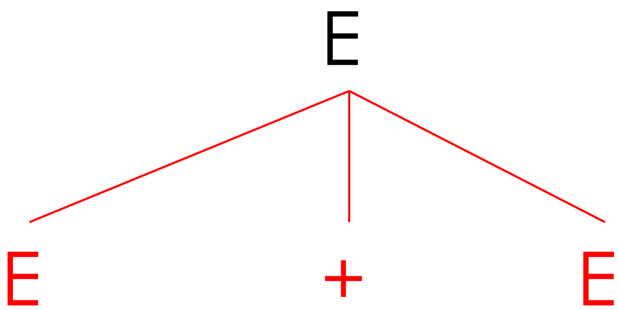
Derivation in Detail (1)

E

E

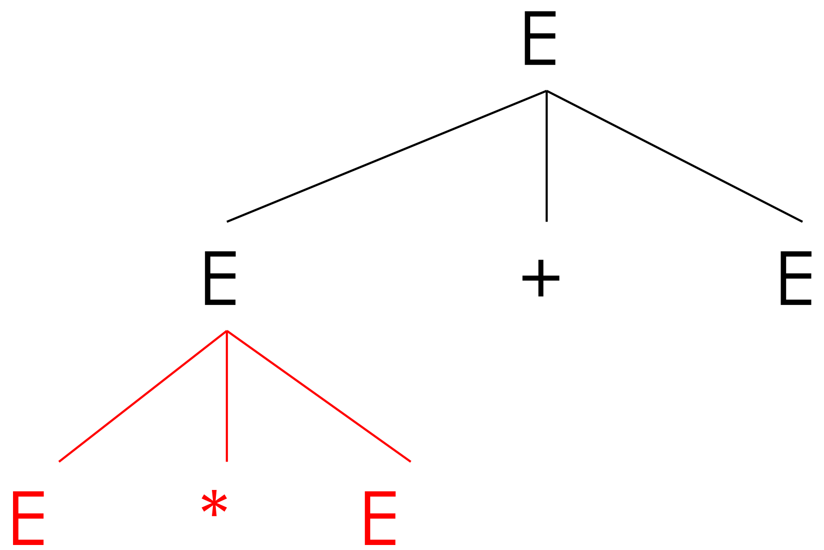
Derivation in Detail (2)

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



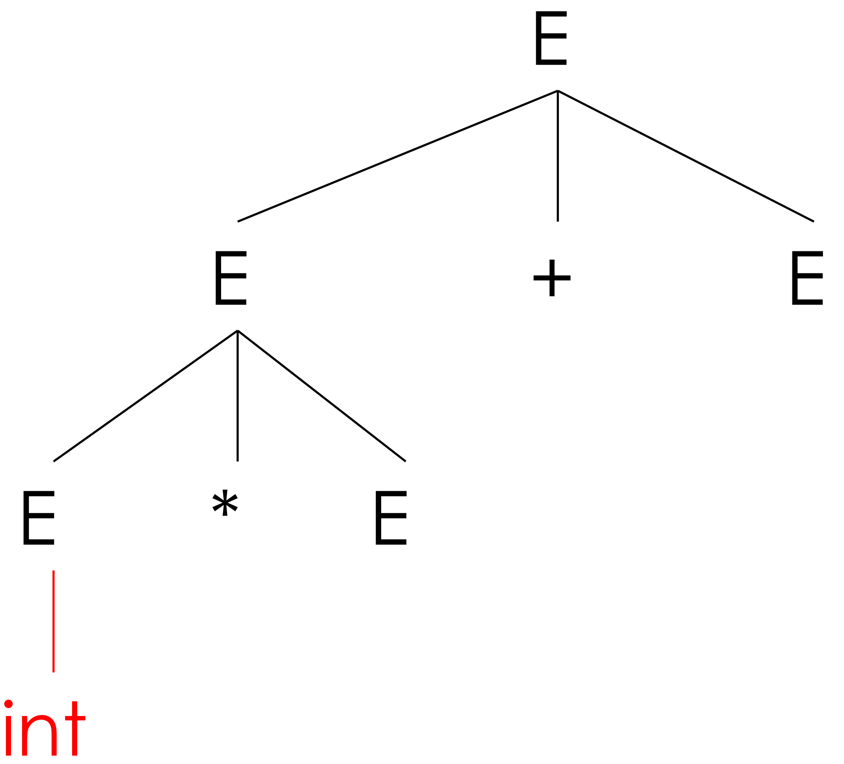
Derivation in Detail (3)

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



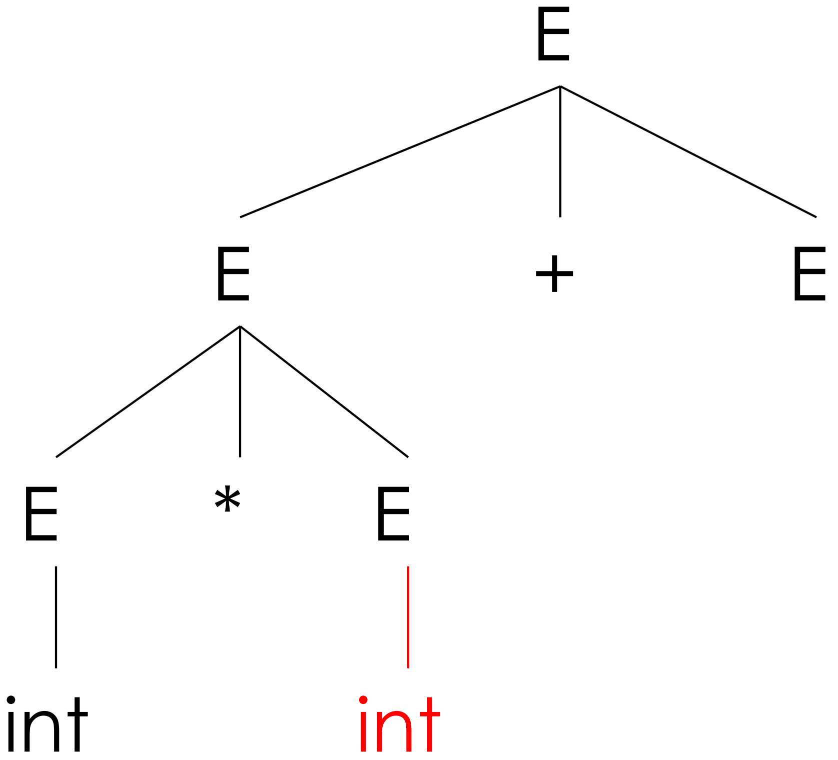
Derivation in Detail (4)

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



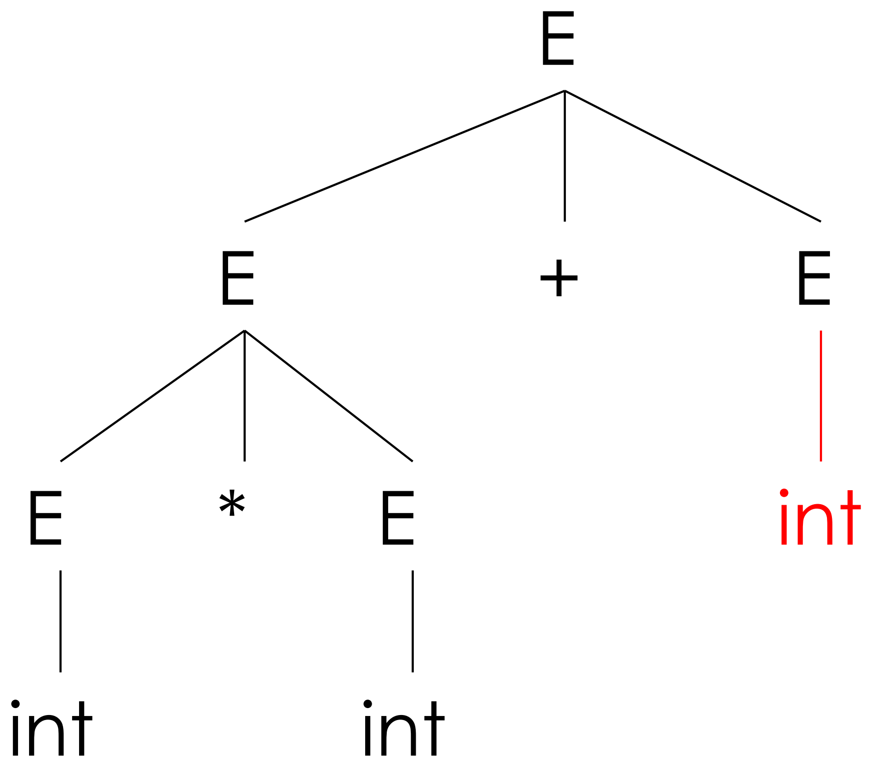
Derivation in Detail (5)

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



Derivation in Detail (6)

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



Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- A in-order 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

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

- 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

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

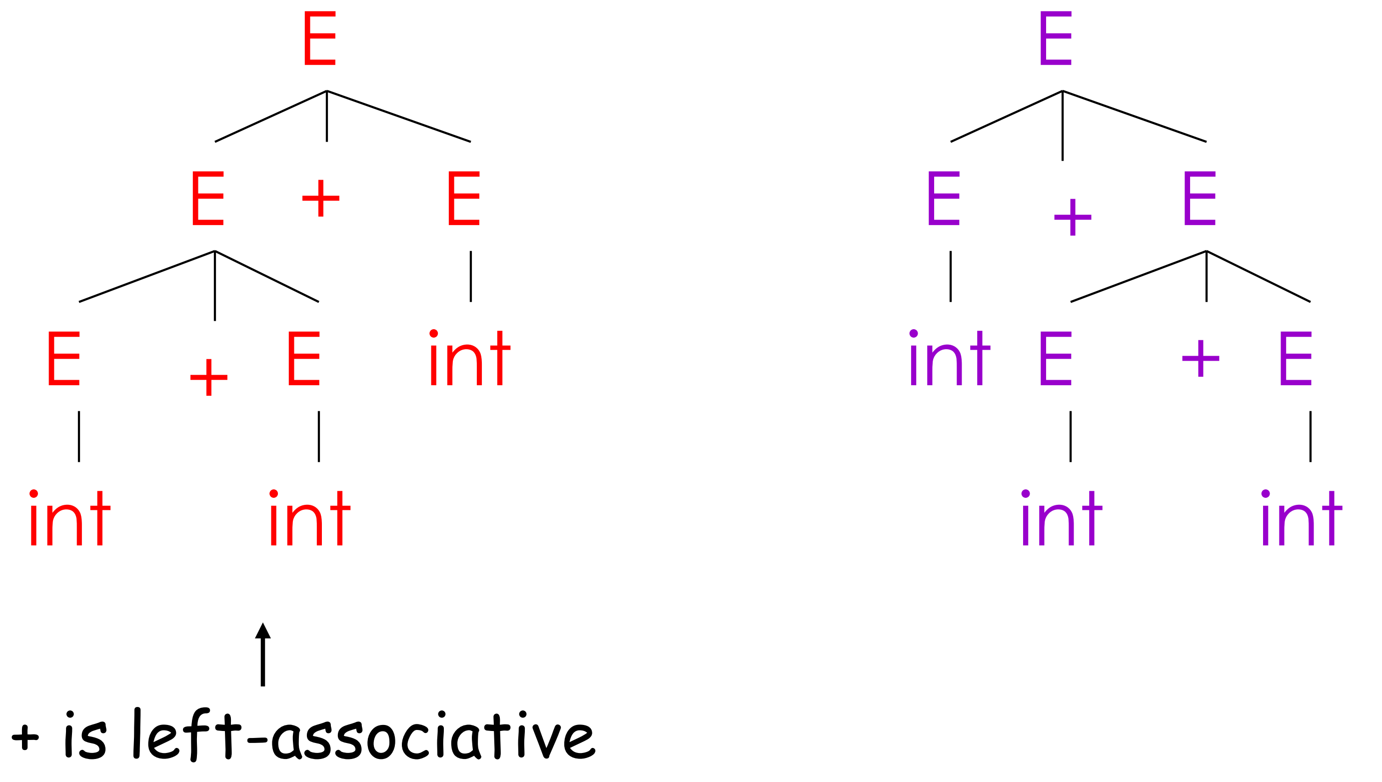
- Strings

int + int + int

int * int + int

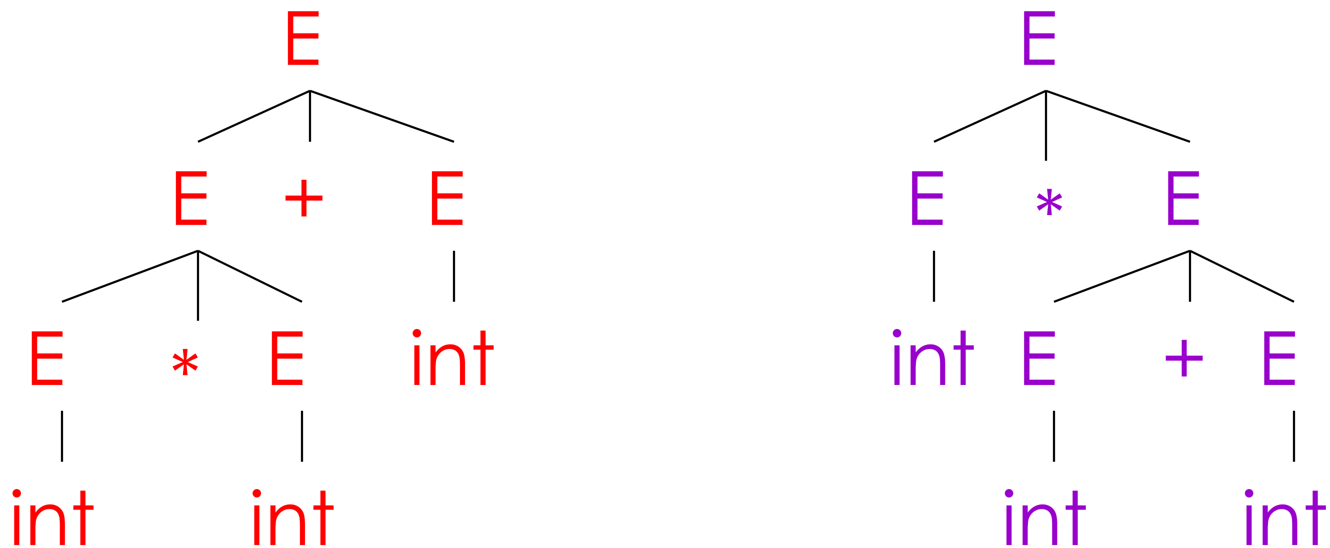
Ambiguity. Example

The string int + int + int has two parse trees



Ambiguity. Example

The string int * int + int has two parse trees



* has higher precedence than +

Ambiguity (Cont.)

- A grammar is *ambiguous* if it has more than one parse tree for some string
  - Equivalently, there is more than one rightmost or leftmost derivation for some string
- Ambiguity is ***BAD***
  - Leaves meaning of some programs ill-defined
- Ambiguity is *common* in programming languages
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity

- Most direct method is to rewrite the grammar unambiguously

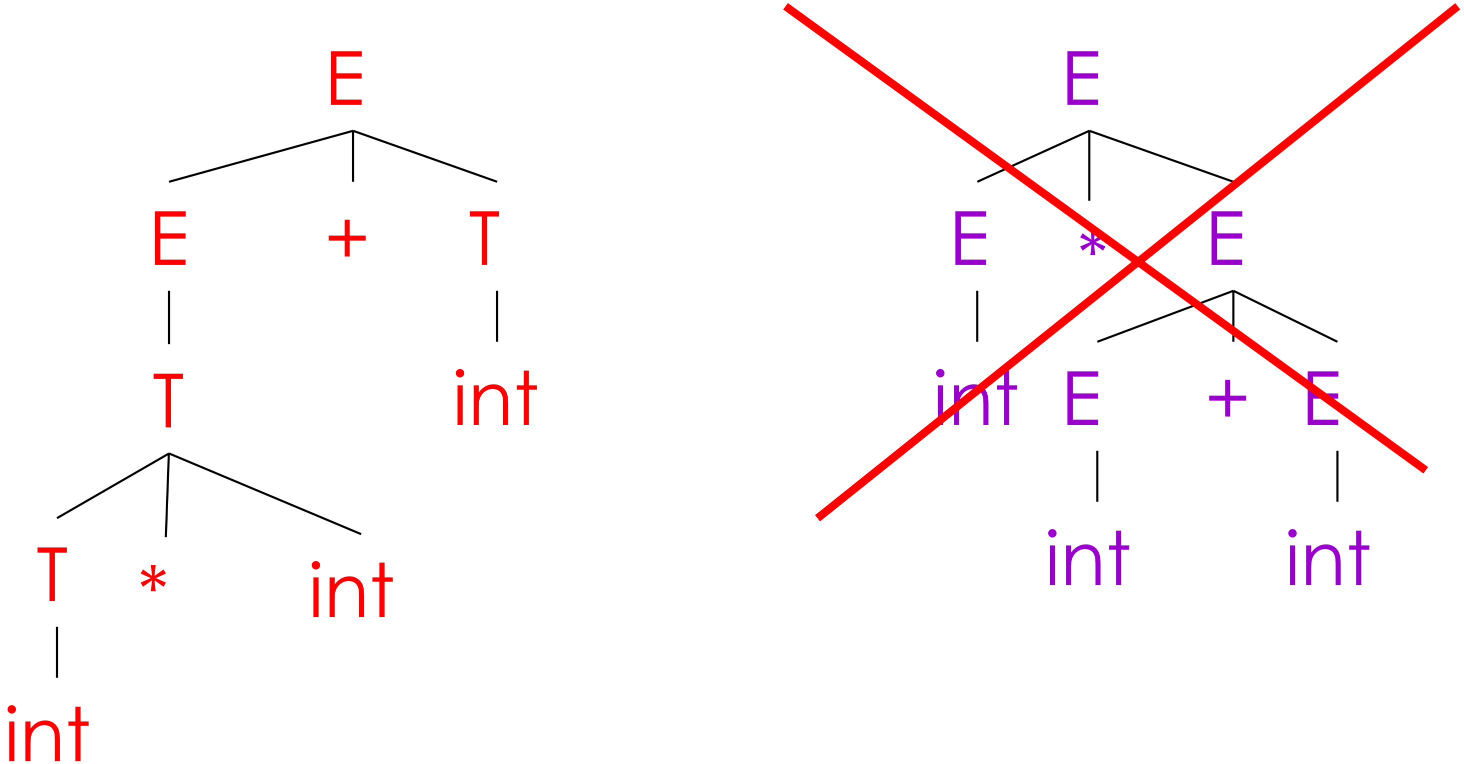
$$E \rightarrow E + T \mid T$$

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

- Enforces precedence of * over +
- Enforces left-associativity of + and *

Ambiguity. Example

The int * int + int has only one parse tree now



Ambiguity: The Dangling Else

- Consider the grammar

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

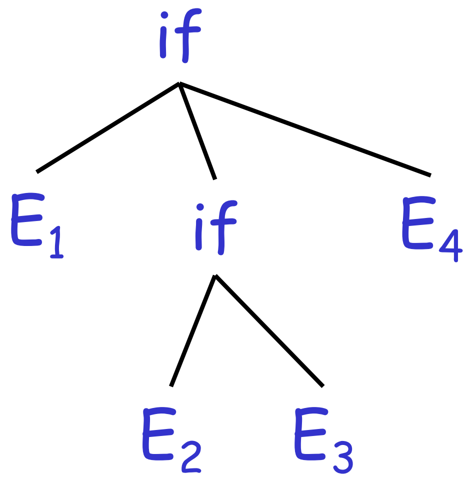
- This grammar is also ambiguous

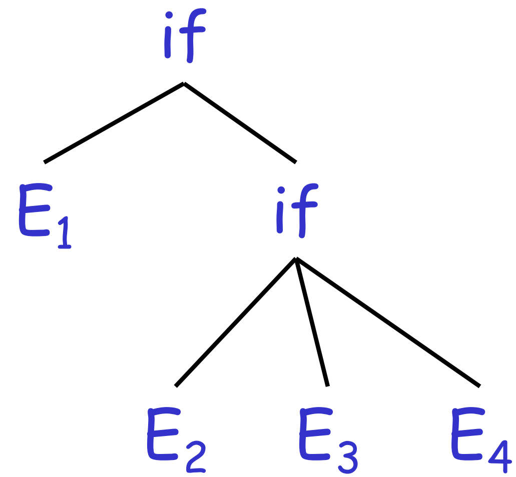
The Dangling Else: Example

- The expression

  $$\text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4$$

  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 (distinguish between matched and unmatched "then")

E $\rightarrow$ MIF /* all then are matched */
| UIF /* some then are unmatched */

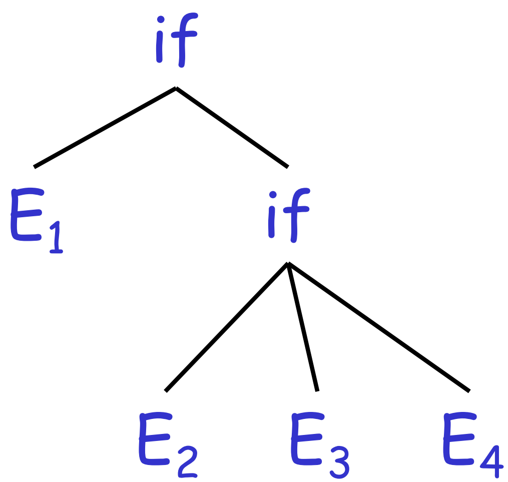
MIF $\rightarrow$ if E then MIF else MIF
| OTHER

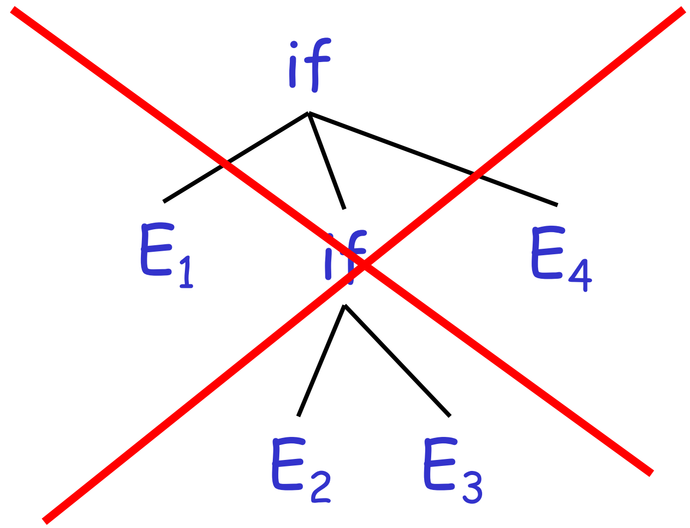
UIF $\rightarrow$ if E then E
| if E then MIF else UIF

- Describes the same set of strings

The Dangling Else: Example Revisited

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





- 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

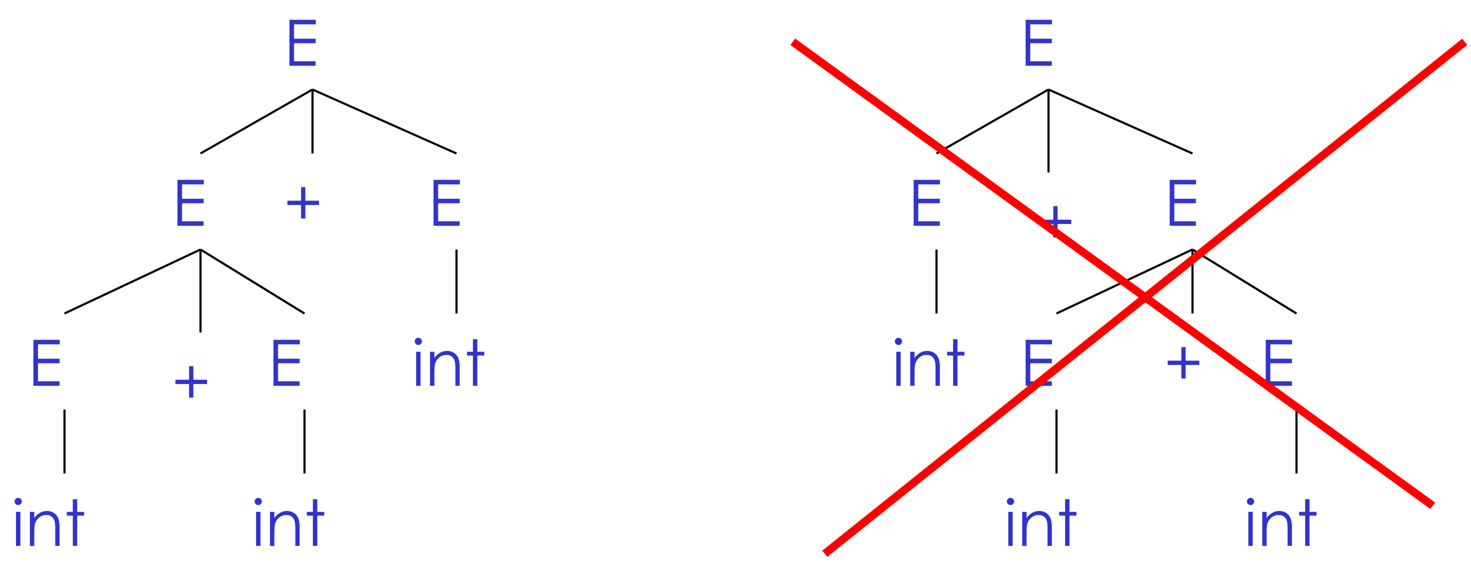
- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations

- Most tools allow *precedence and associativity declarations* to disambiguate grammars

- Examples …

Associativity Declarations

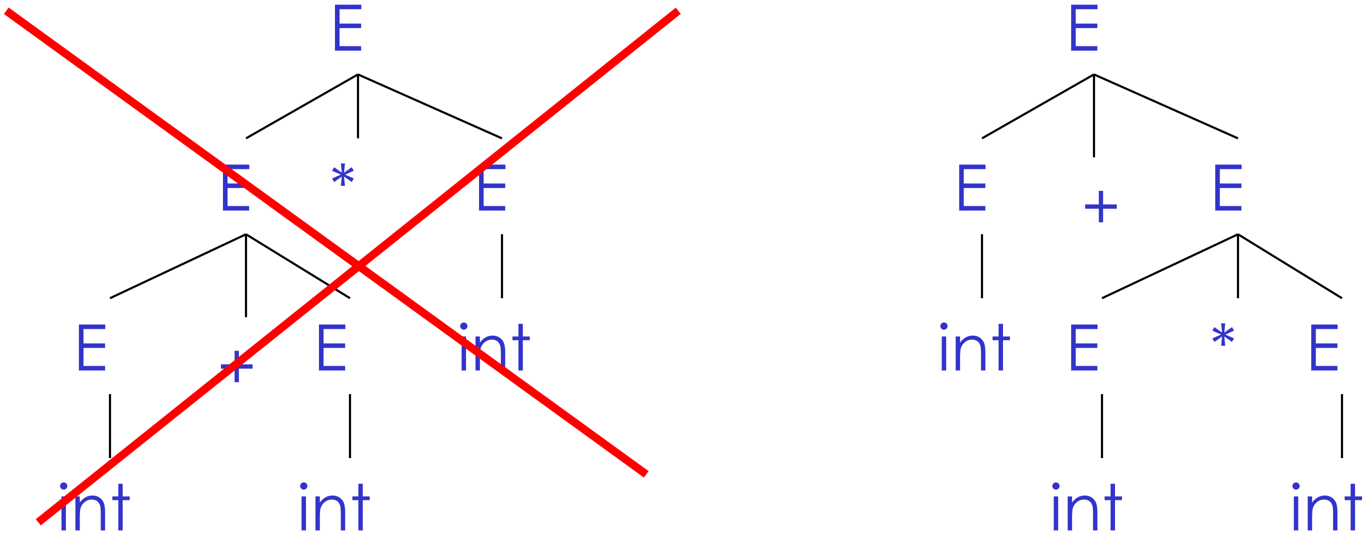
- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of $int + int + int$



- Left-associativity declaration: %left '+'

Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid E * E \mid int$
  - And the string int + int * int



- Precedence declarations: %left '+'
  %left '*'