450 COMPILERS

COMPUTER SCIENCE

News & Info

- Who's Hiring May 2016
  - https://news.ycombinator.com/item?id=11611867

- SoCal Code Camp | San Diego, CA 6/25-6/26
  - http://www.socalcodecamp.com/

Administrivia

- Lab 05
  - Due Thursday

Review

- Context-Free Grammar (Grammar)
  - Called a Production
  - Four Components
    - 1) Set of tokens, terminals
    - 2) A set of nonterminals
    - 3) A set of Productions
    - 4) Nonterminal Start Symbol

Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages

- Regular Languages
  - The weakest formal languages widley used
  - Many applications

Beyond Regular Languages

- Many languages are not regualr

- Strings of balance parantheses are not regular:
- $\{ (^i)^i \mid i > 0\}$

What can Regular Languages Express?

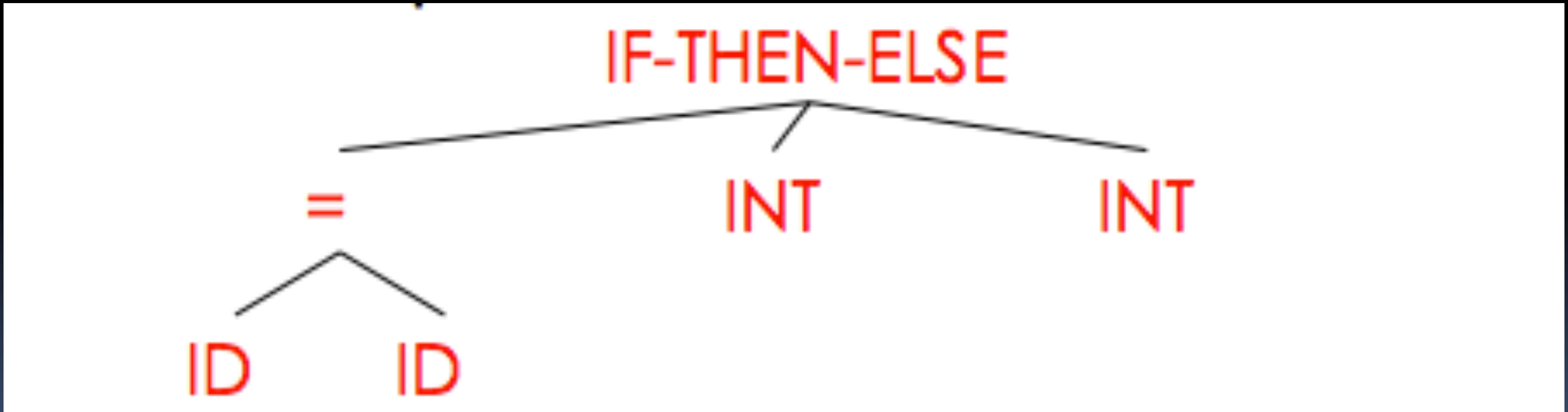
- Languages requiring counting modulo a fixed integer
- Intuition: A finite automaton that runs long enough must repeat states
- Finite automaton can't remember # of times it has visited a particular state

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

- Cflat:
  - if x = y then 1 else 2
- Parser Input
  - IF ID = ID THEN INT ELSE INT
- Parser Output



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 language constructs have recursive structure
- An EXPR is
  - If EXPR then EXPR else EXPR
  - while EXPR loop
- CFG are a natural notation for this recursive structure

CFGs continued

- 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

Examples of CFGs

- EXPR -> if EXPR then EXPR else EXPR
- | while EXPR loop EXPR
- | id

Examples of CFGs continued

Simple arithmetic expressions:

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

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 CGF

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

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:

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

Terminals

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

Arithmetic Example

Simple arithmetic expressions:

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

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 (ex. Bison)

Notes

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar
  - Note: Tools for regular languages (ex. Flex) are sensitive to the form of the regular expressions, but this is rarely a problem in practice

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \to \dots \to \dots$$

A derivation can be drawn as a tree

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

Derivation Example

- Grammar

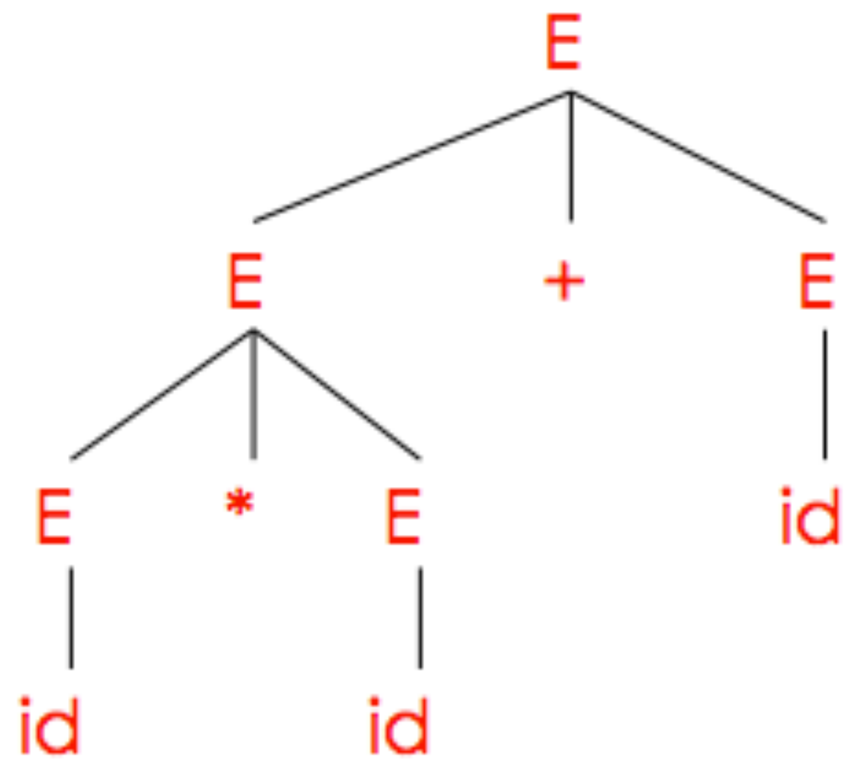
$$E \rightarrow E{+}E \mid E * E \mid (E) \mid id$$

- String

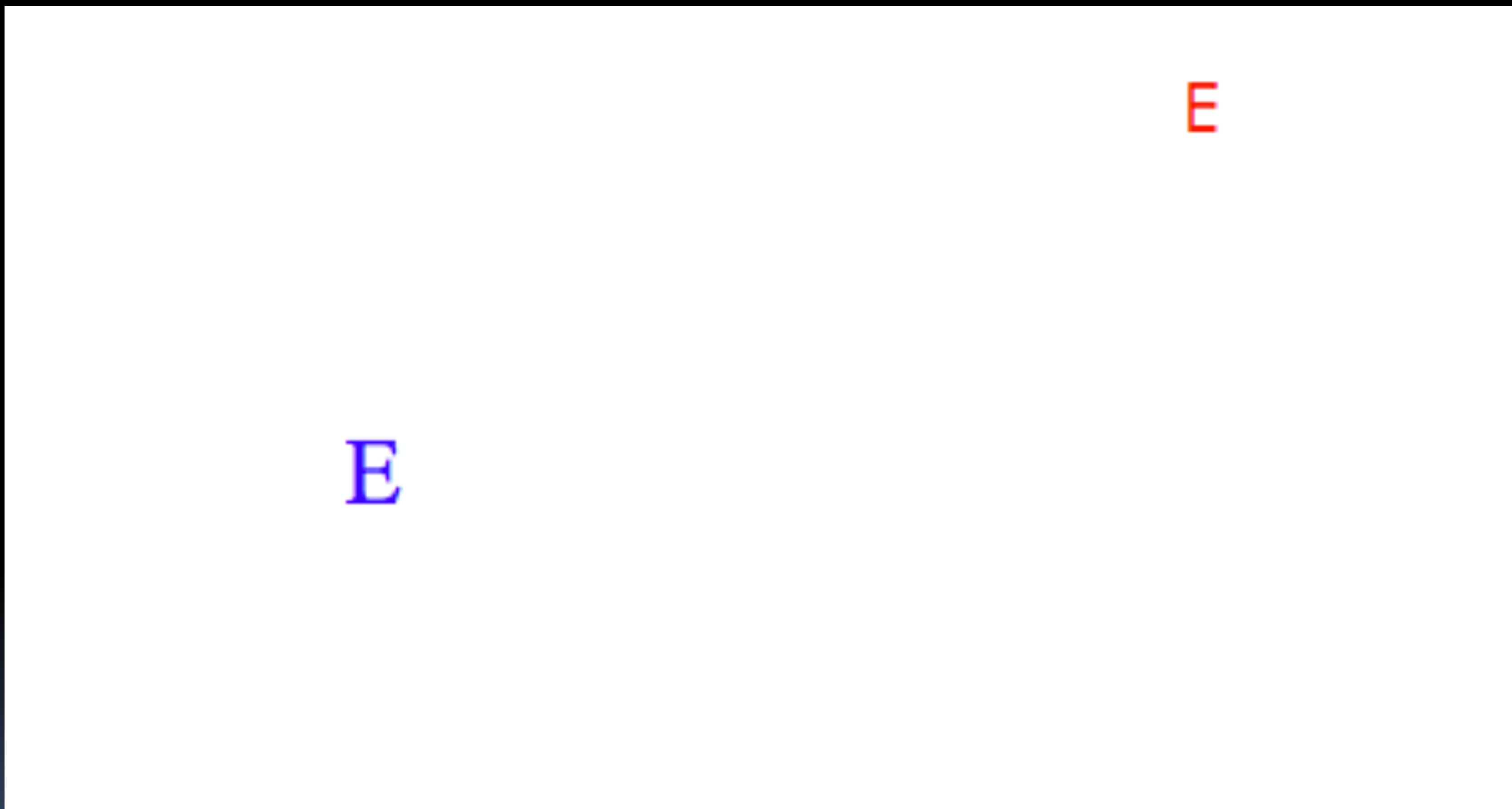
$$id \; * \; id + id$$

Derivation Example

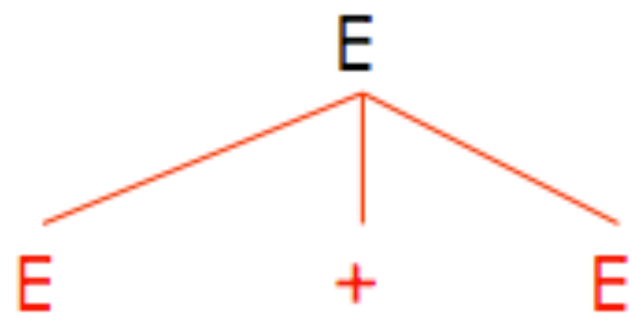
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Derivation in Detail (1)



Derivation in Detail (2)

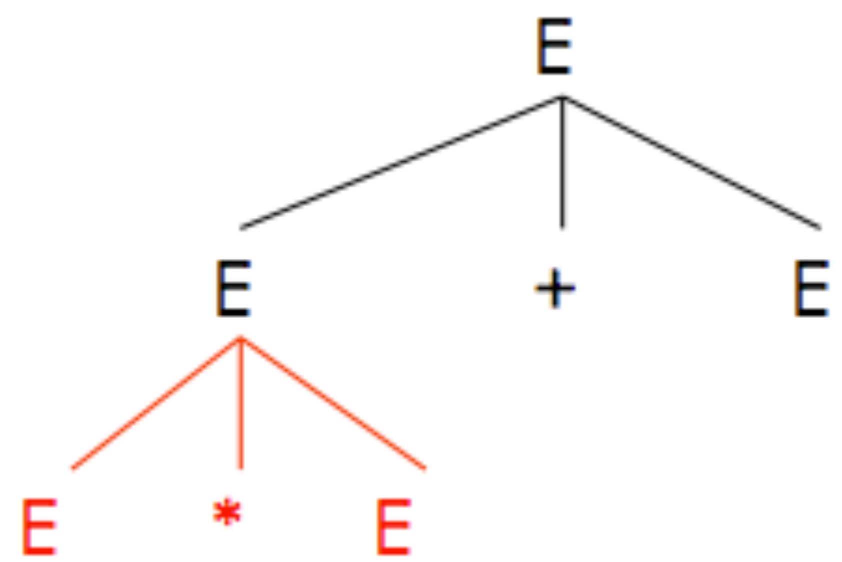


E

$\rightarrow$ E+E

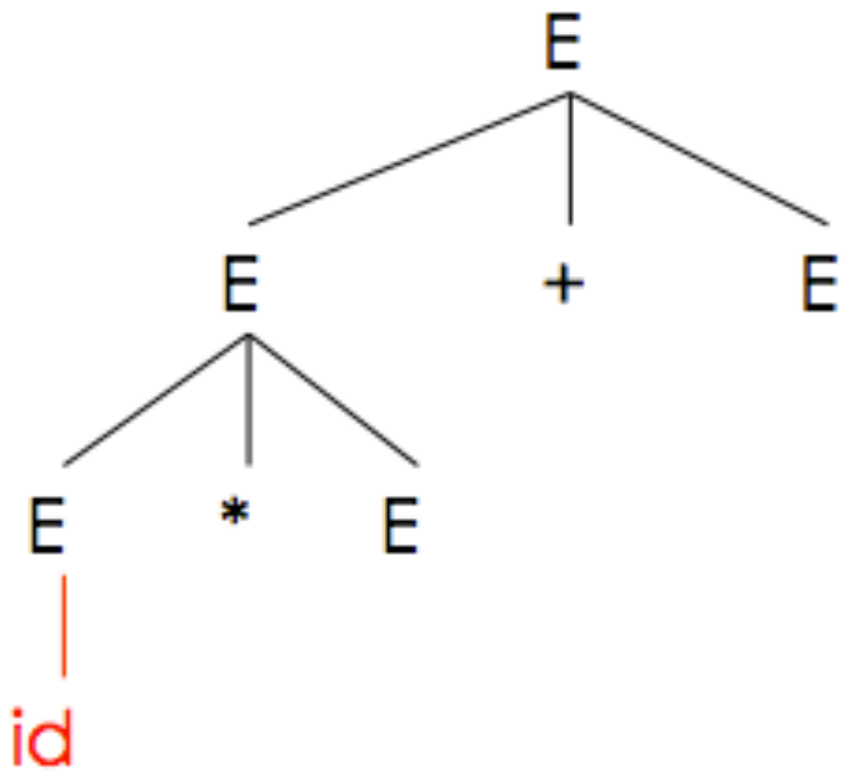
Derivation in Detail (3)

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



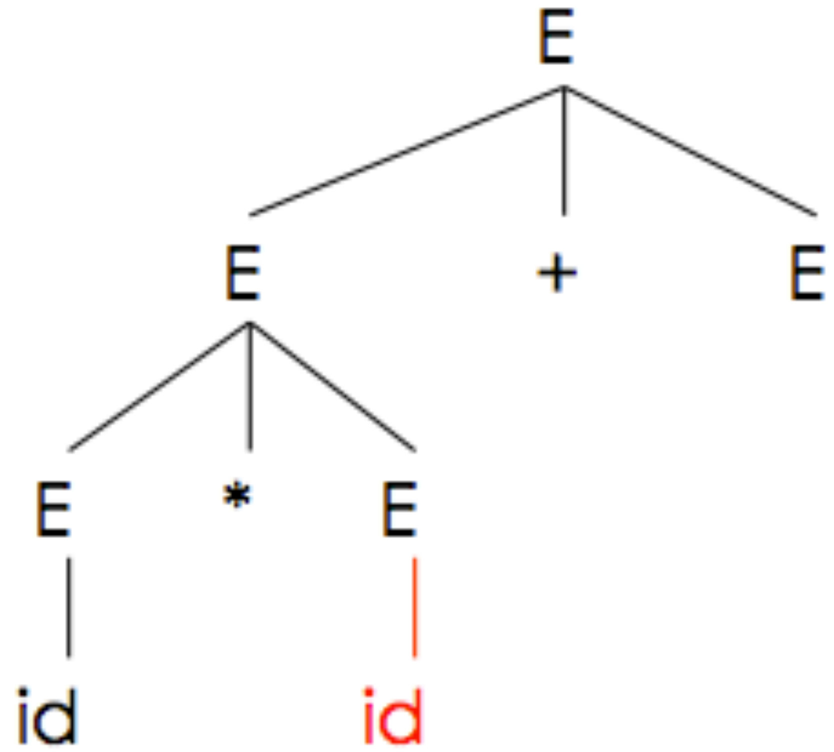
Derivation in Detail (4)

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



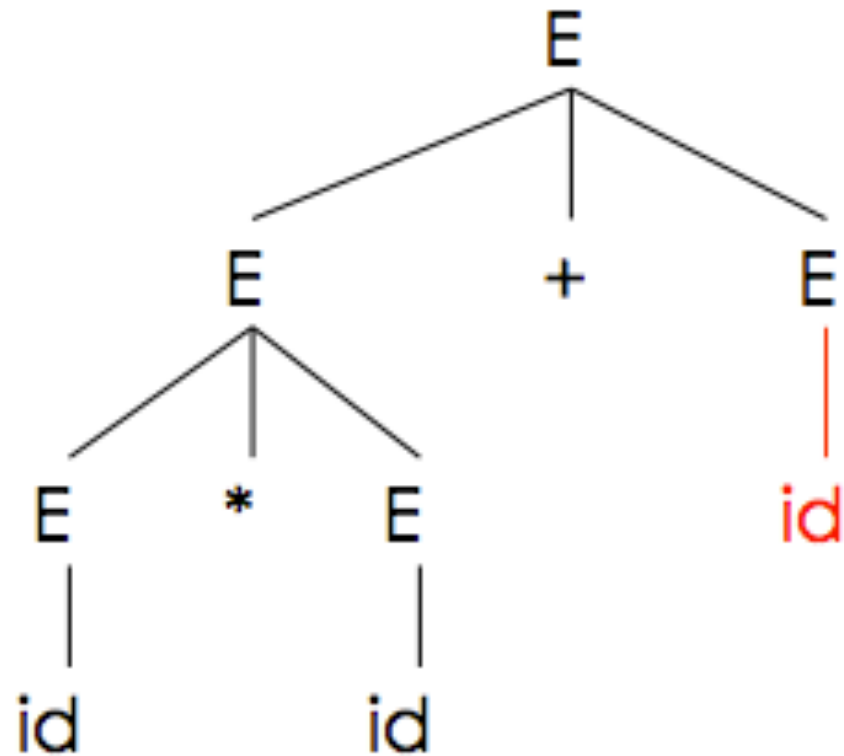
Derivation in Detail (5)

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



Derivation in Detail (6)

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



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

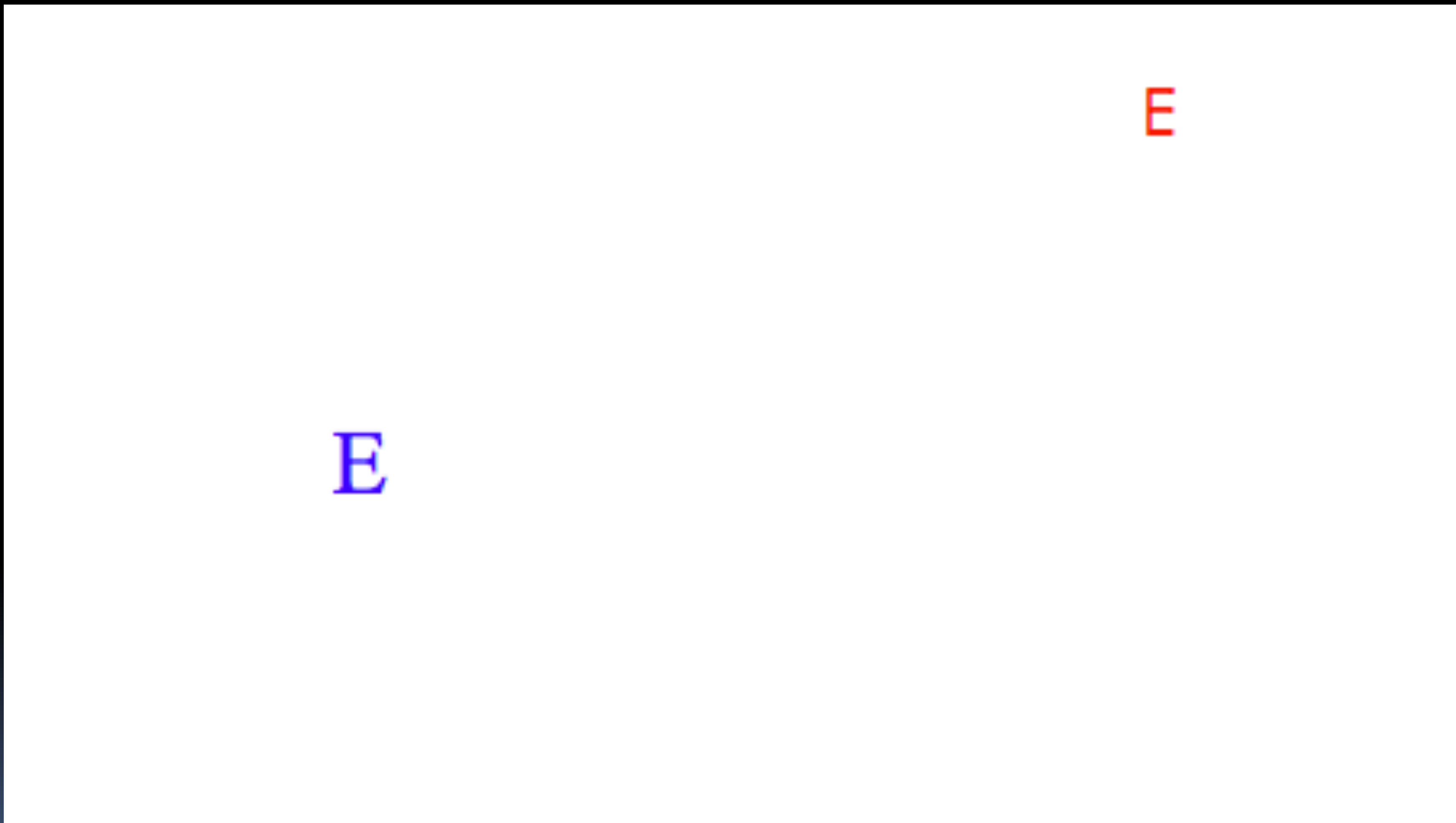
- The parse tree shows the association of operations, the input string does not

Left-most and Right-most Derivations

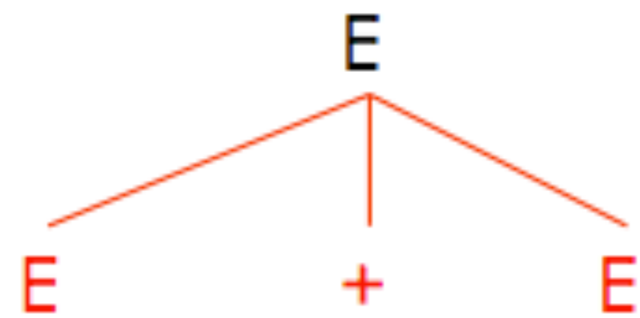
- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

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

Right-most Derivation in Detail (1)



Right-most Derivation in Detail (2)



E

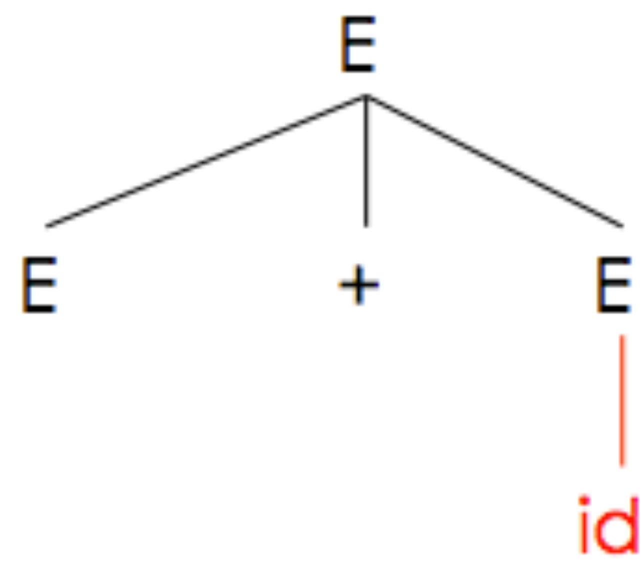
→ E+E

Right-most Derivation in Detail (3)

E

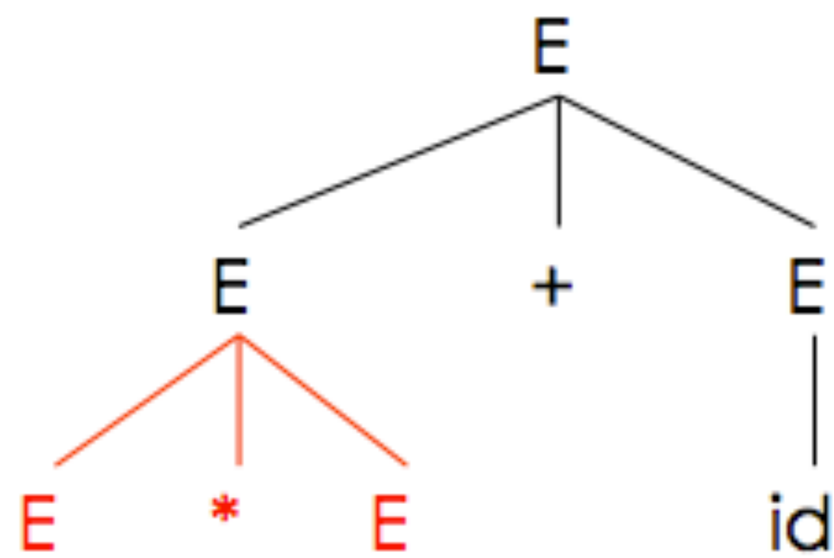
$\rightarrow E+E$

$\rightarrow E+id$



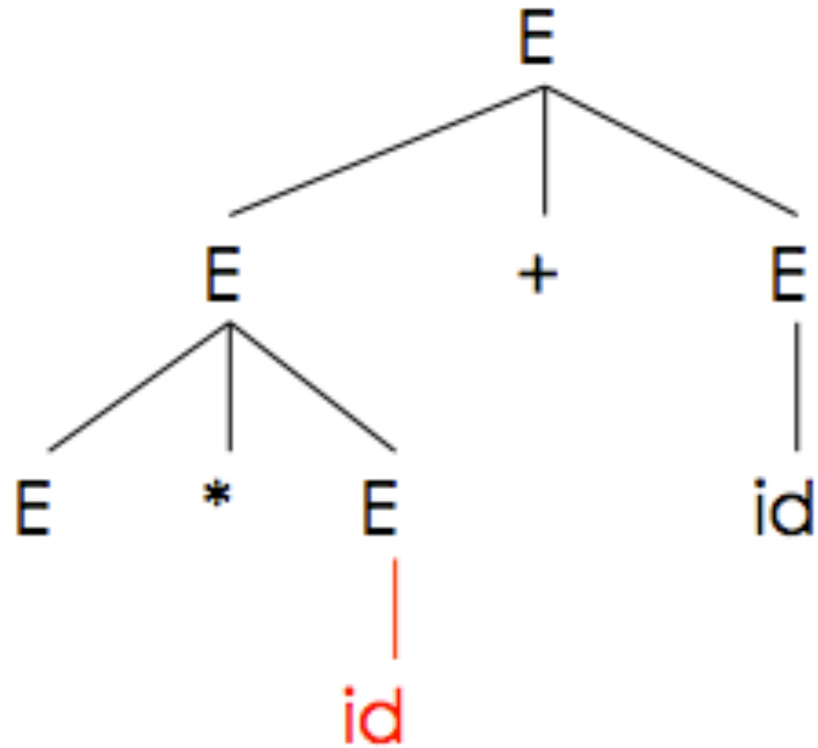
Right-most Derivation in Detail (4)

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



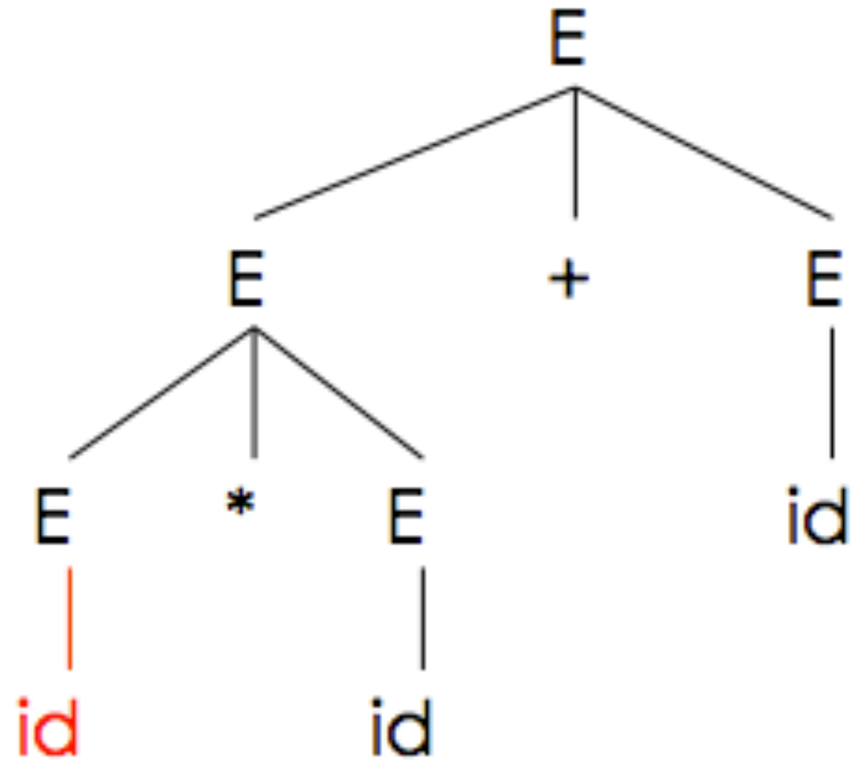
Right-most Derivation in Detail (5)

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



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

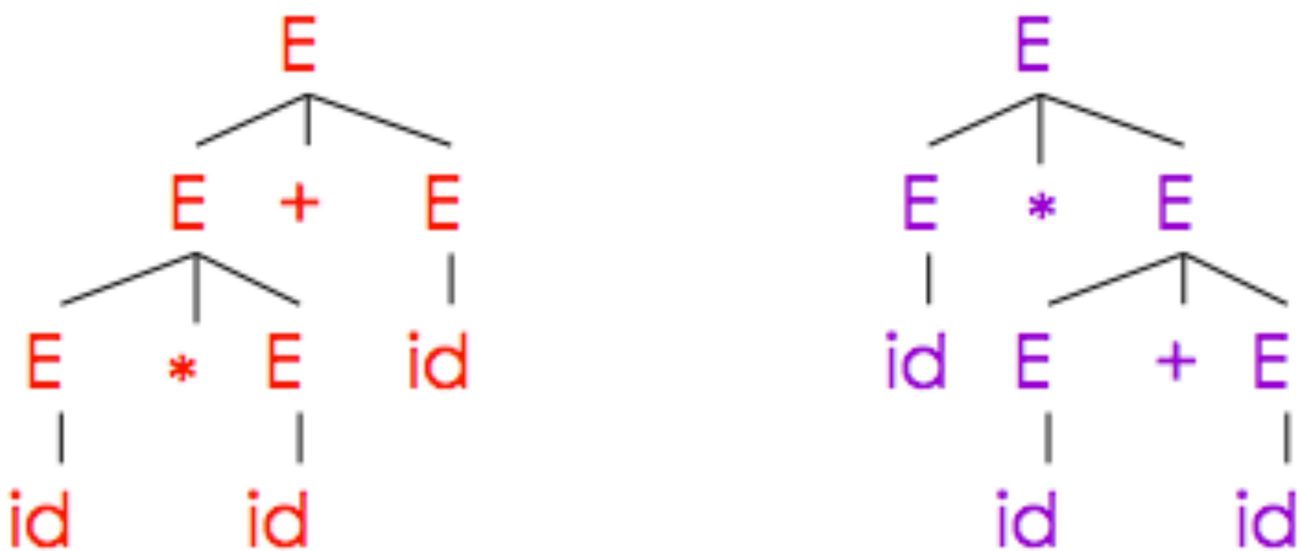
- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Ambiguity

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

Ambiguity continued

This string has two parse trees



Ambiguity continued

- 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

$$E \rightarrow E' + E \mid E'$$

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

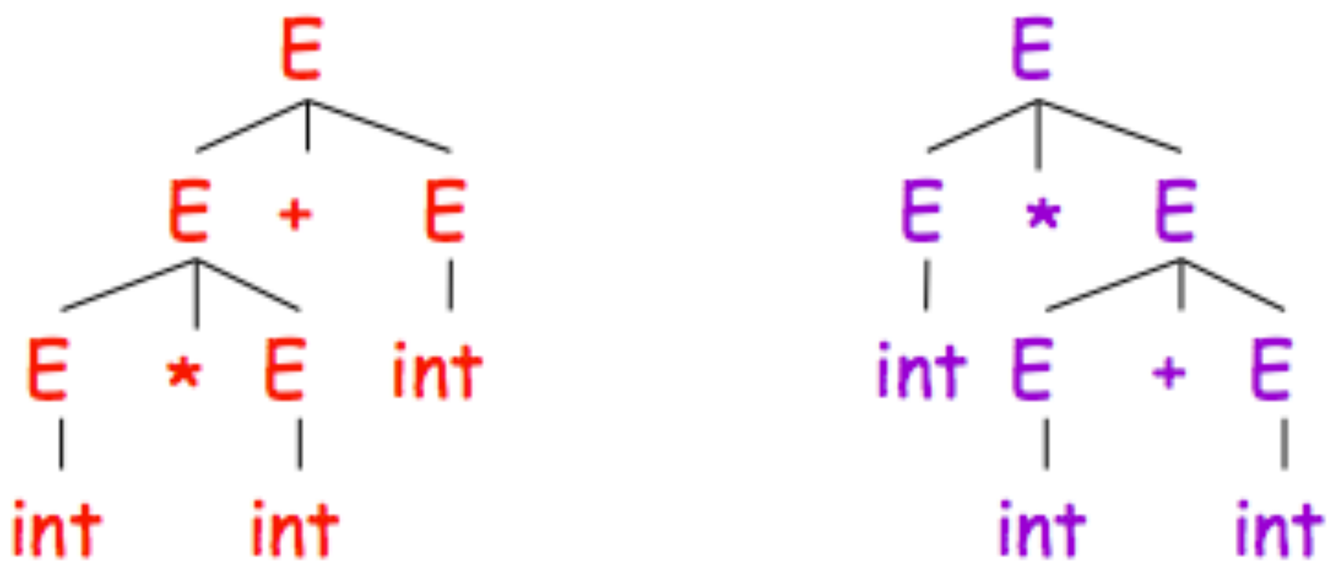
- Enforces precedence of * over +

Ambiguity in Arithmetic Expressions

- Recall the grammar

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

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



Ambiguity: The Dangling Else

- Consider the grammar

$$E \rightarrow \text{if } E \text{ then } E \mid \text{if } E \text{ then } E \text{ else } E \mid \text{OTHER}$$

- 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

The Dangling Else: Fix

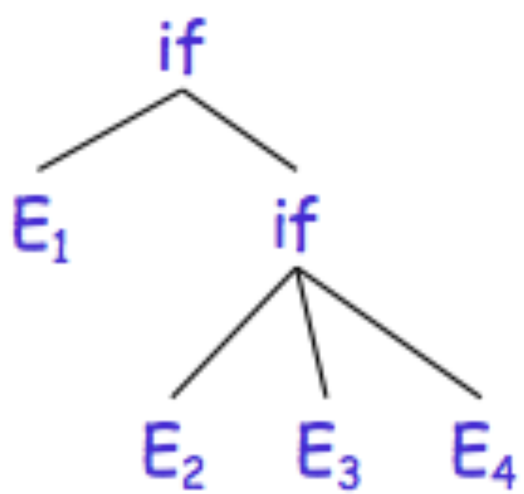
- else matches the closest unmatched then
- We can describe this in the grammar

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

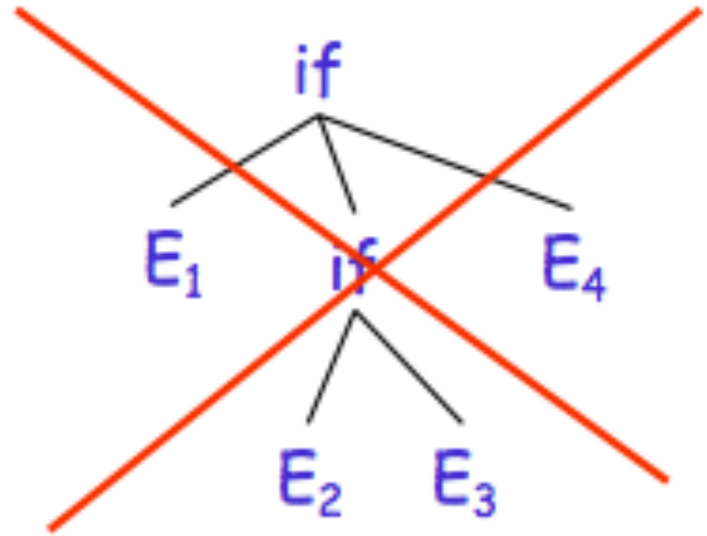
- Describes the same set of strings

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 amiguous grammar to an unambiguous one

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

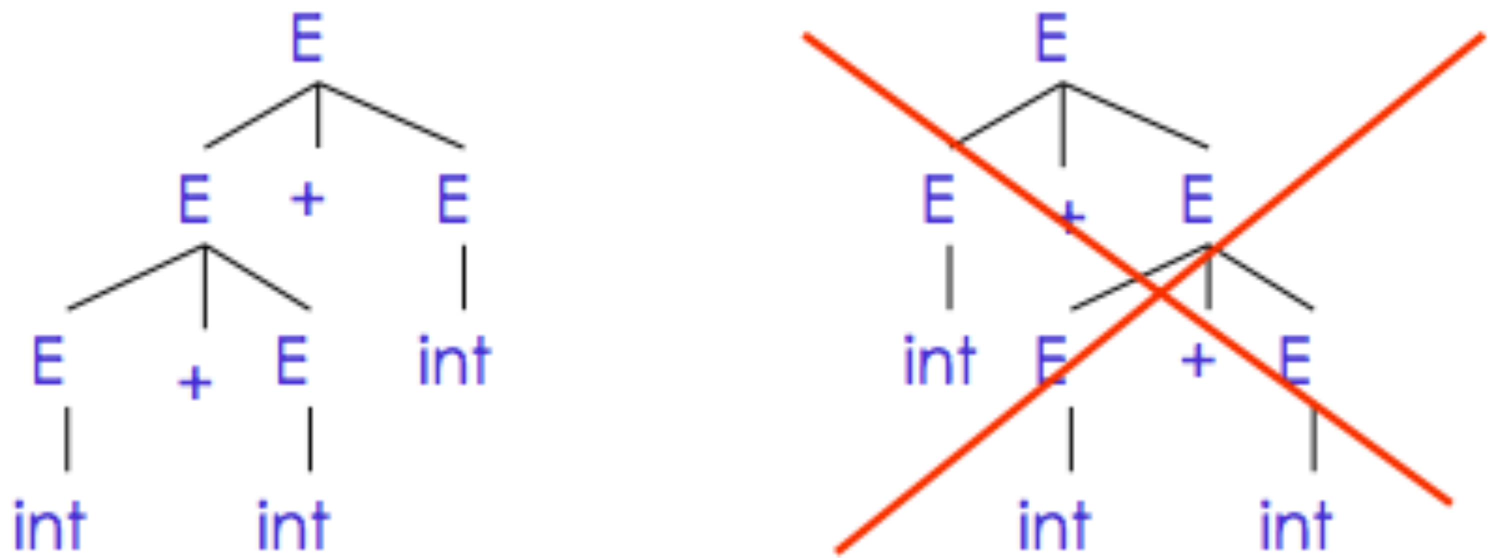
Precedence and Associativity Declaration

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

- Most tolls allow precedence and associativity declarations to disambiguate grammars
- Examples….

Associativity Declarations

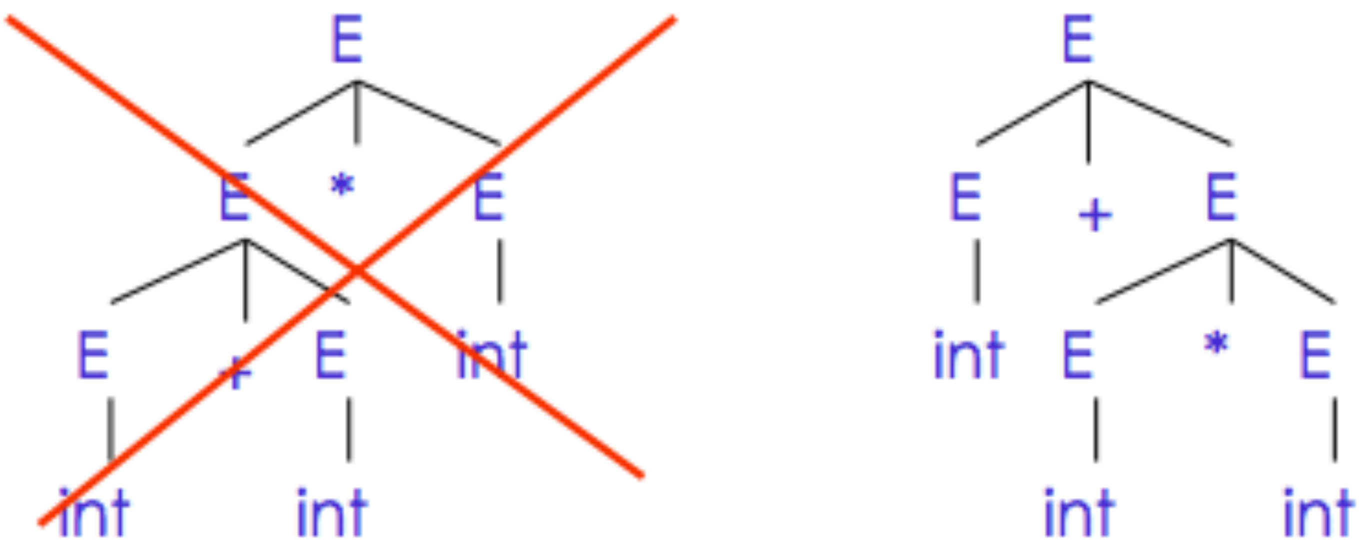
- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int



- 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 *