Introduction to Parsing
Ambiguity and Syntax Errors

Outline

- Regular languages revisited
- Parser overview
- Context-free grammars (CFG's)
- Derivations
- Ambiguity
- Syntax errors

Languages and Automata

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

- A finite automaton *cannot remember* number of times it has visited a particular state
- because a finite automaton has finite memory
  - Only enough to store in which state it is
  - Cannot count, except up to a finite limit
- Many languages are not regular
- E.g., the language of balanced parentheses is not regular: $\{ (^i\ )^i \mid i \geq 0\}$

The Functionality of the Parser

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

Example

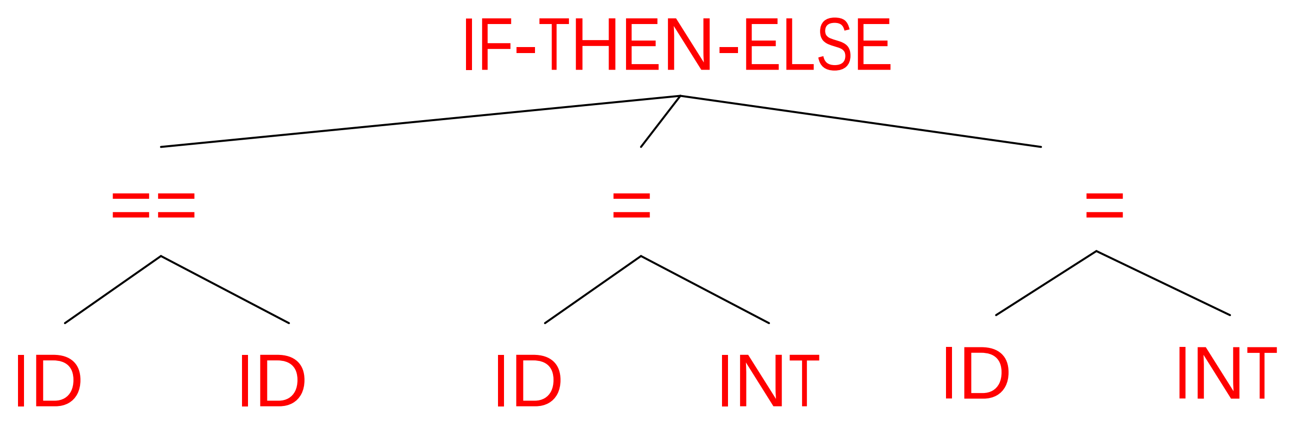
- If-then-else statement

```
if (x == y) then z = 1; else z = 2;
```

- Parser input

IF (ID == ID) THEN ID = INT; ELSE ID = INT;

- Possible 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

- Many programming language constructs have a recursive structure

- E.g. A STMT is of the form

  if COND then STMT else STMT , or

  while COND do STMT , 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 $X \in N$ the productions are of the form

$X \to \varepsilon$, or

$X \to Y_1 Y_2 \dots Y_n$ 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

Example: A small fragment of our language:

$$
\begin{aligned}
\text{STMT} \rightarrow\ & \text{if COND then STMT else STMT} \\
|\ & \text{while COND do STMT} \\
|\ & \text{id = int}
\end{aligned}
$$

One More Example

Grammar for simple arithmetic expressions:

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

The Language of a CFG

Read productions as replacement rules:

$X \rightarrow Y_1 \ldots Y_n$

Means X can be replaced by $Y_1 \ldots Y_n$ (in this order)

$X \rightarrow \varepsilon$

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 the right-hand side of some production

$$X \rightarrow Y_1 \dots Y_n$$

(3) Repeat (2) until there are no non-terminals in the string

The Language of a CFG (Cont.)

More formally, we write

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

if there is a production

$$X_i \rightarrow Y_1 \cdots Y_m$$

We write

$$X_1 \cdots X_n \xrightarrow{*} Y_1 \cdots Y_m$$

if

$$X_1 \cdots X_n \rightarrow \cdots \rightarrow \cdots \rightarrow Y_1 \cdots Y_m$$

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:

$$\left\{ a_1 \ldots a_n \mid S \xrightarrow{*} a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \right\}$$

Terminals

- Terminals are called so 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 the CFG G

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

Two equivalent ways of writing the grammar G:

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

or

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

Example

A fragment of our example language (simplified):

$$
\begin{aligned}
\text{STMT} &\to \text{if COND then STMT} \\
&\mid \text{if COND then STMT else STMT} \\
&\mid \text{while COND do STMT} \\
&\mid \text{id = int} \\
\text{COND} &\to \text{(id == id)} \\
&\mid \text{(id != id)}
\end{aligned}
$$

Example (Cont.)

Some elements of the our language

id = int

if (id == id) then id = int else id = int

while (id != id) do id = int

while (id == id) do while (id != id) do id = int

if (id != id) then if (id == id) then id = int else id = int

Arithmetic Example

Simple arithmetic expressions:

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

Some elements of the language:

id	$\mathrm{id} + \mathrm{id}$
(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 just "yes" or "no"; we also need the parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG's
  - e.g., yacc/bison/ML-yacc/...

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \rightarrow \cdots \rightarrow \cdots \rightarrow \cdots$$

A derivation can be drawn as a tree

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

Derivation Example

- Grammar

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

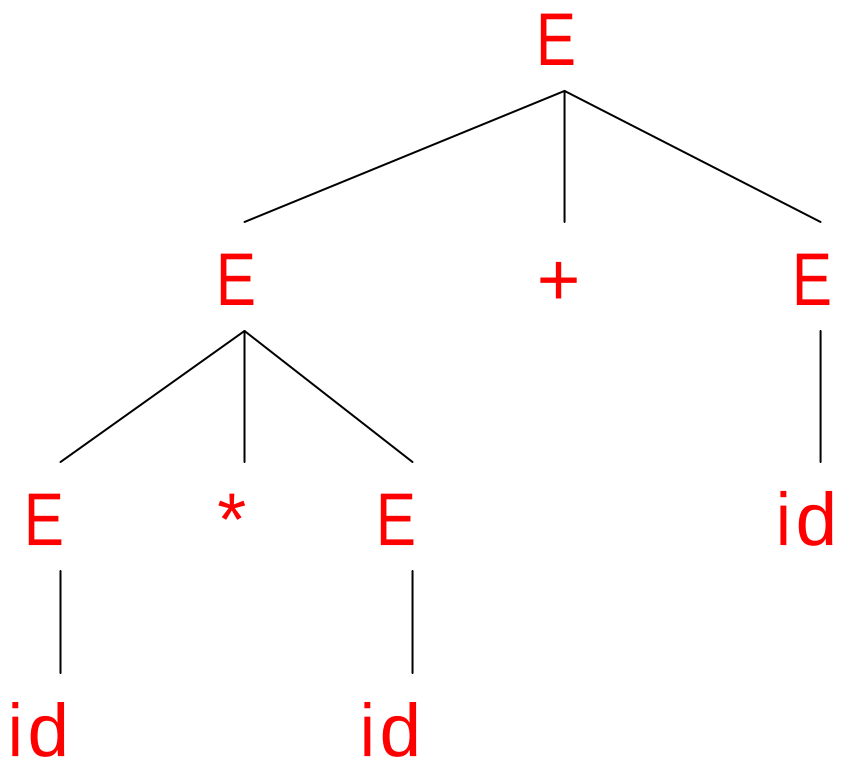
- String

$$id * id + id$$

Derivation Example (Cont.)

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

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



Derivation in Detail (1)

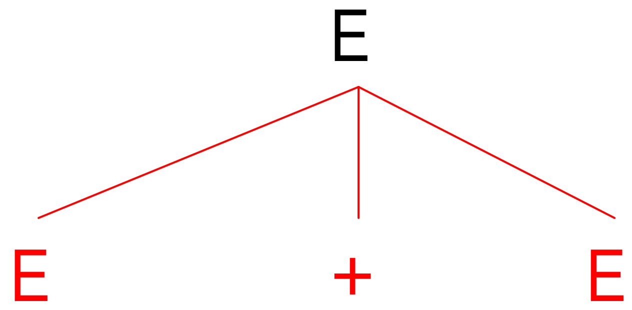
$E \rightarrow E+E \mid E*E \mid (E) \mid id$

E

E

Derivation in Detail (2)

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

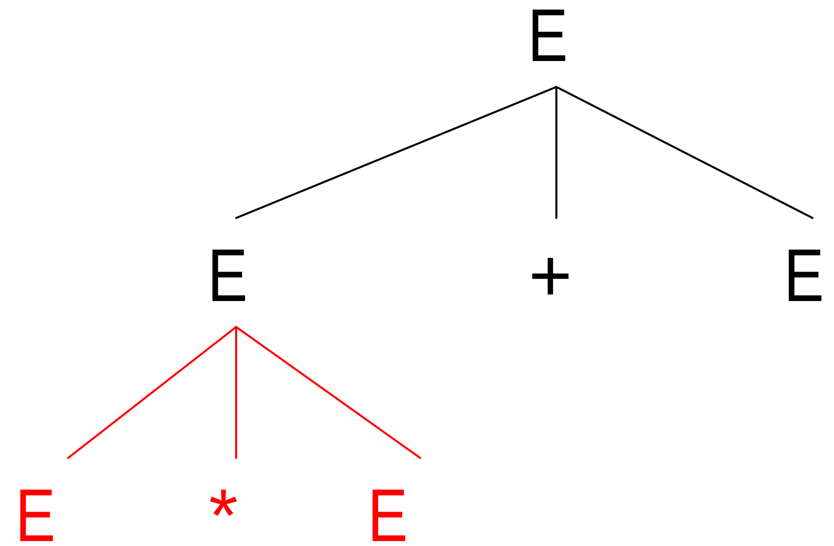


$$E$$

$$\rightarrow \quad E+E$$

Derivation in Detail (3)

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

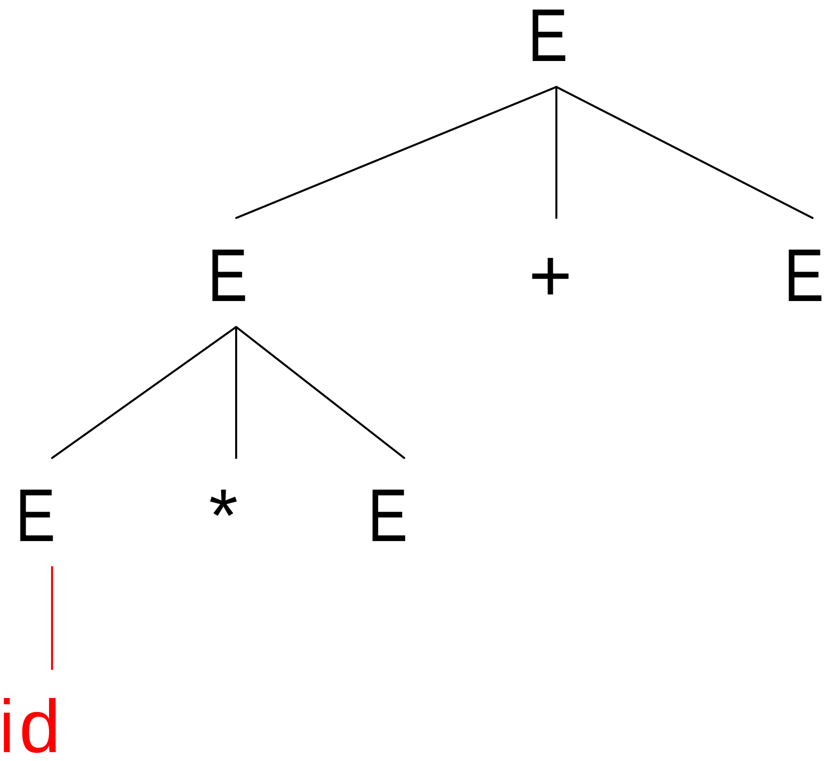


$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E
\end{aligned}
$$

Derivation in Detail (4)

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

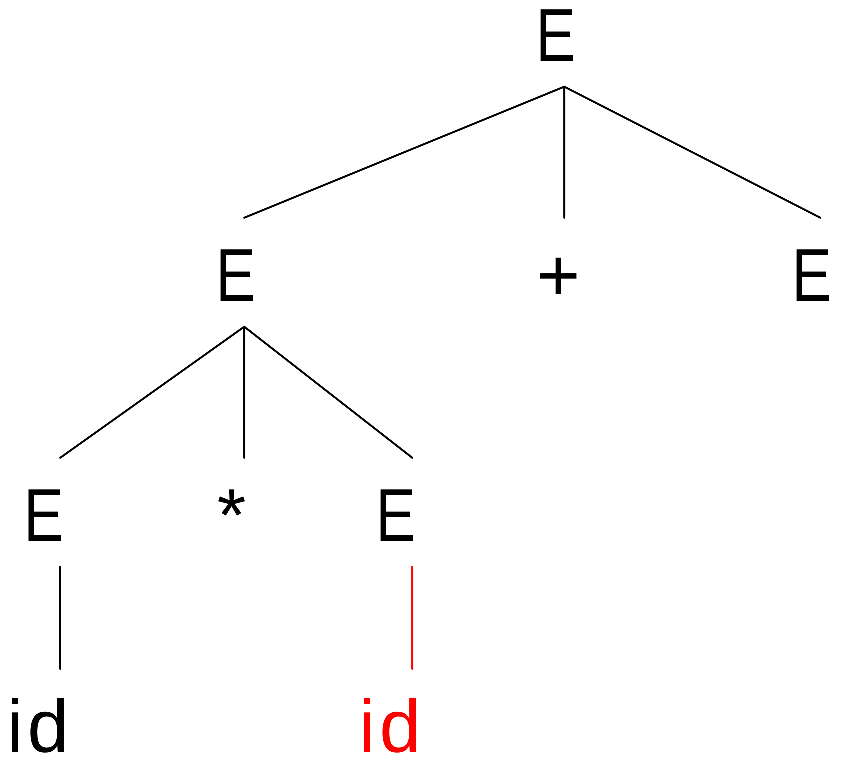
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E
\end{aligned}
$$



Derivation in Detail (5)

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

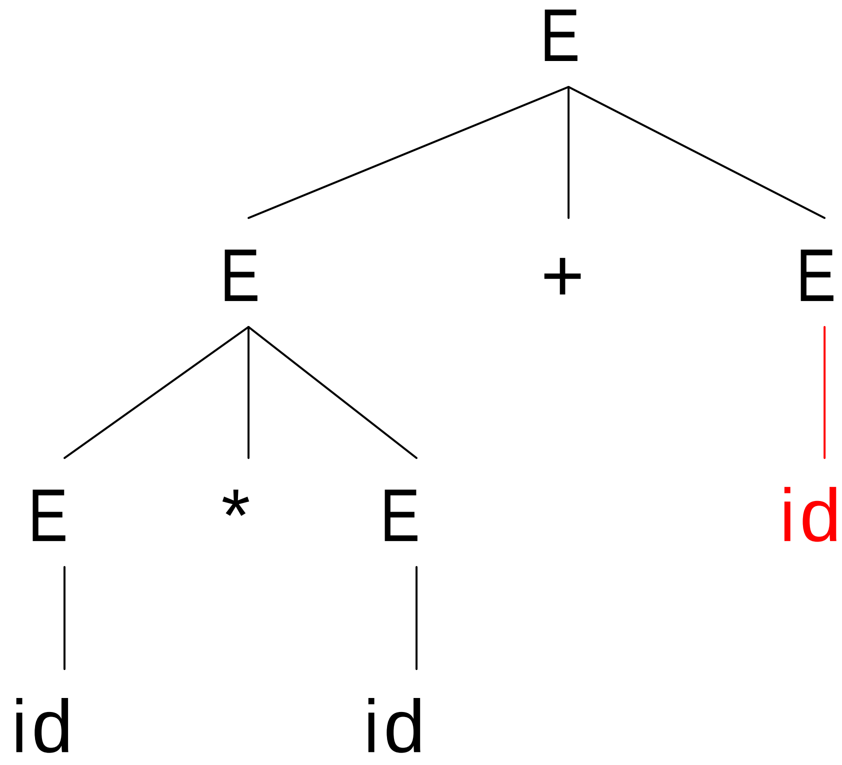
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E
\end{aligned}
$$



Derivation in Detail (6)

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

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



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

- What was shown before was a *left-most derivation*
  - At each step, we replaced the left-most non-terminal
- There is an equivalent notion of a *right-most derivation*
  - Shown on the right

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

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

Right-most Derivation in Detail (1)

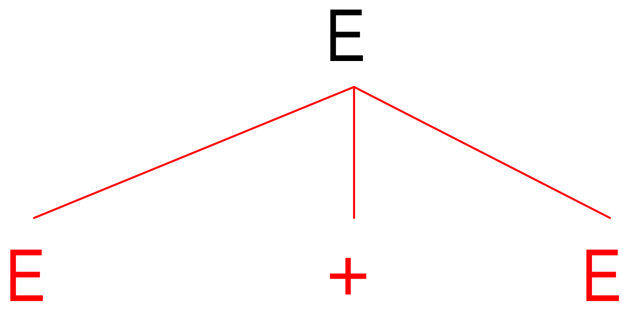
$E \rightarrow E+E \mid E*E \mid (E) \mid id$

E

E

Right-most Derivation in Detail (2)

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



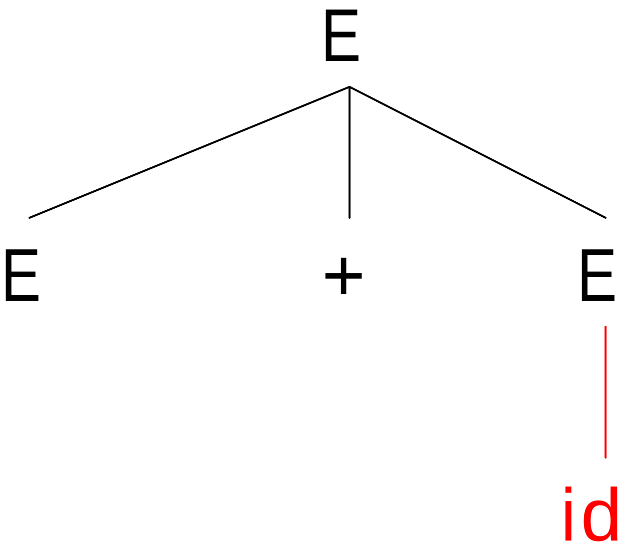
E

$\rightarrow \quad E+E$

Right-most Derivation in Detail (3)

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

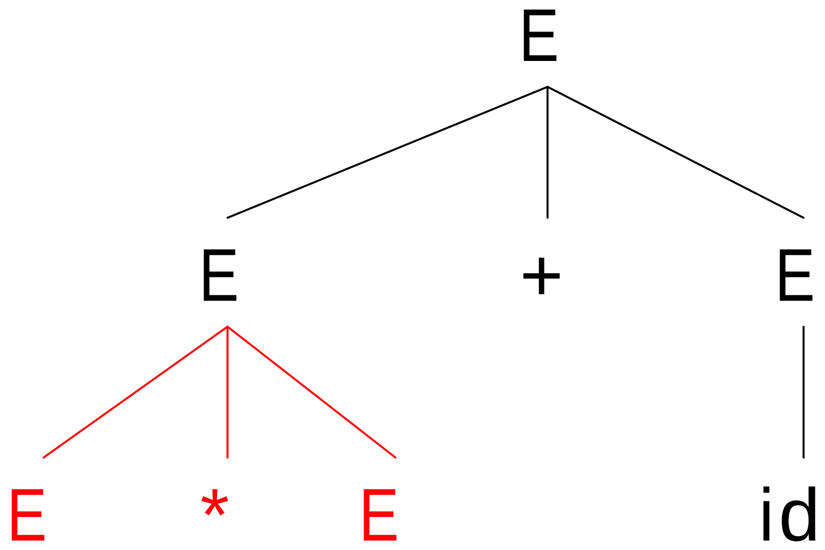
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id
\end{aligned}
$$



Right-most Derivation in Detail (4)

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

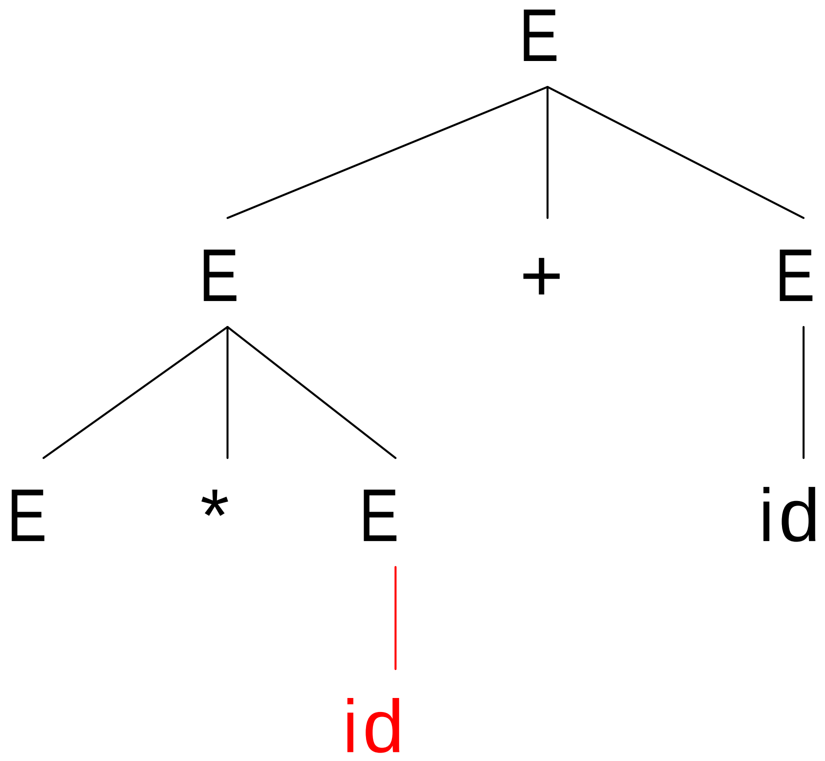
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id
\end{aligned}
$$



Right-most Derivation in Detail (5)

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

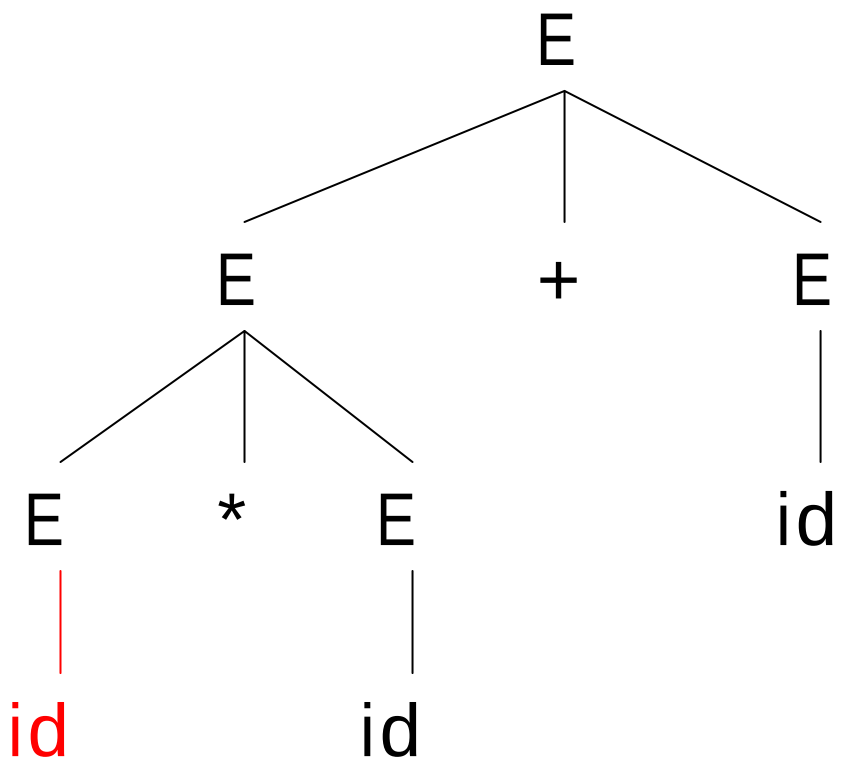
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id
\end{aligned}
$$



Right-most Derivation in Detail (6)

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

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



Derivations and Parse Trees

- Note that:
  - right-most and left-most derivations have the same parse tree
  - for each parse tree, there is a right-most and a left-most derivation

- The difference *is just in 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

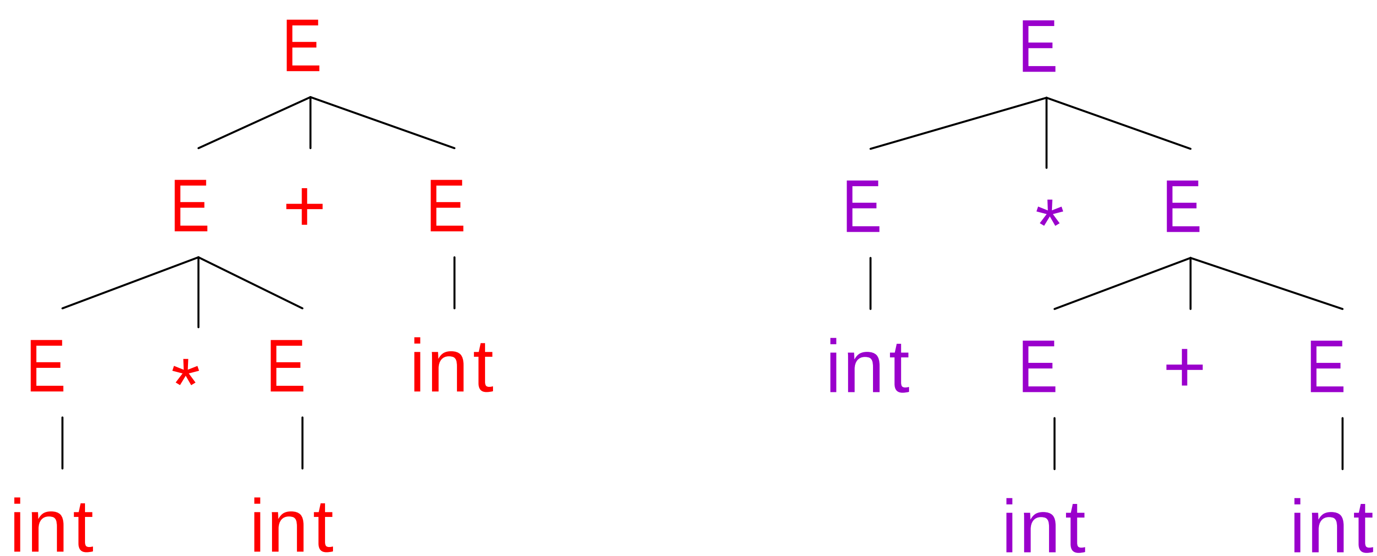
- Left-most and right-most derivations are important in parser implementation

Ambiguity

- Grammar:

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

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



Ambiguity (Cont.)

- A grammar is *ambiguous* if it has more than one parse tree for some string
  - Equivalently, if there is more than one right-most or left-most derivation for some string
- Ambiguity is <u>bad</u> in programming languages
  - Leaves meaning of some programs ill-defined
- Ambiguity is <u>common</u> 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

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

- This grammar enforces precedence of * over +

Ambiguity: The Dangling Else

- Consider the following grammar

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

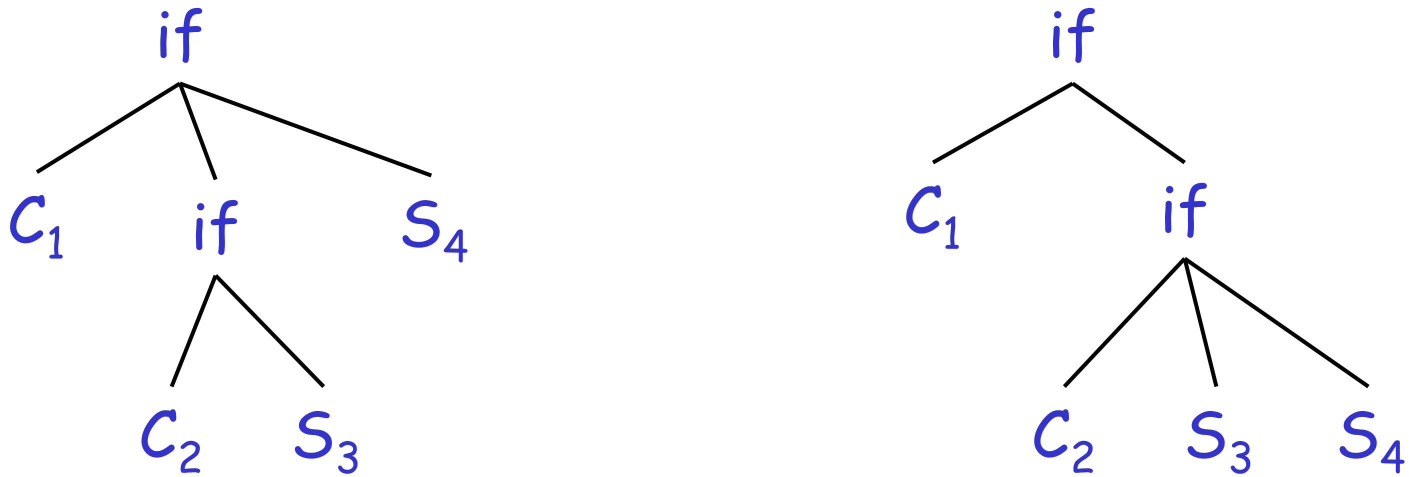
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  $\text{if } C_1 \text{ then if } C_2 \text{ then } S_3 \text{ else } S_4$

  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

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

S → MIF /* all then are matched */
  | UIF /* some then are unmatched */

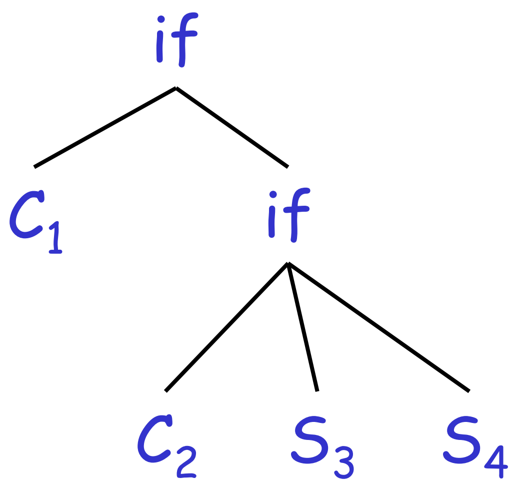
MIF → if C then MIF else MIF
  | OTHER

UIF → if C then S
  | if C then MIF else UIF

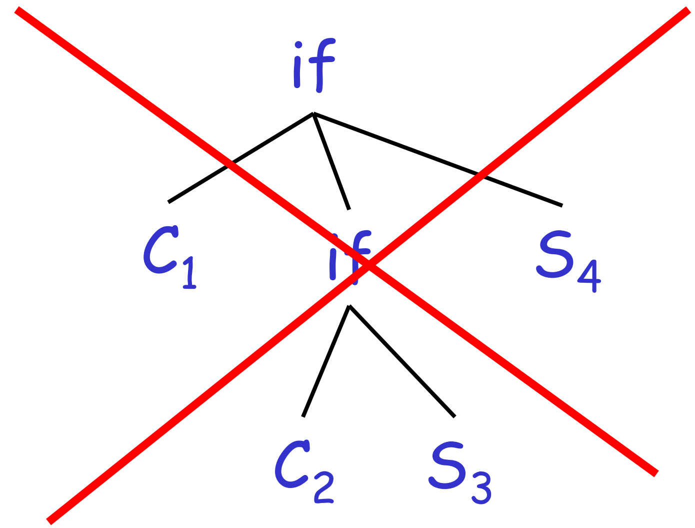
- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if C_1 then if C_2 then S_3 else S_4



- A valid parse tree (for a UIF)



- Not valid because the then expression is not a MIF

Ambiguity

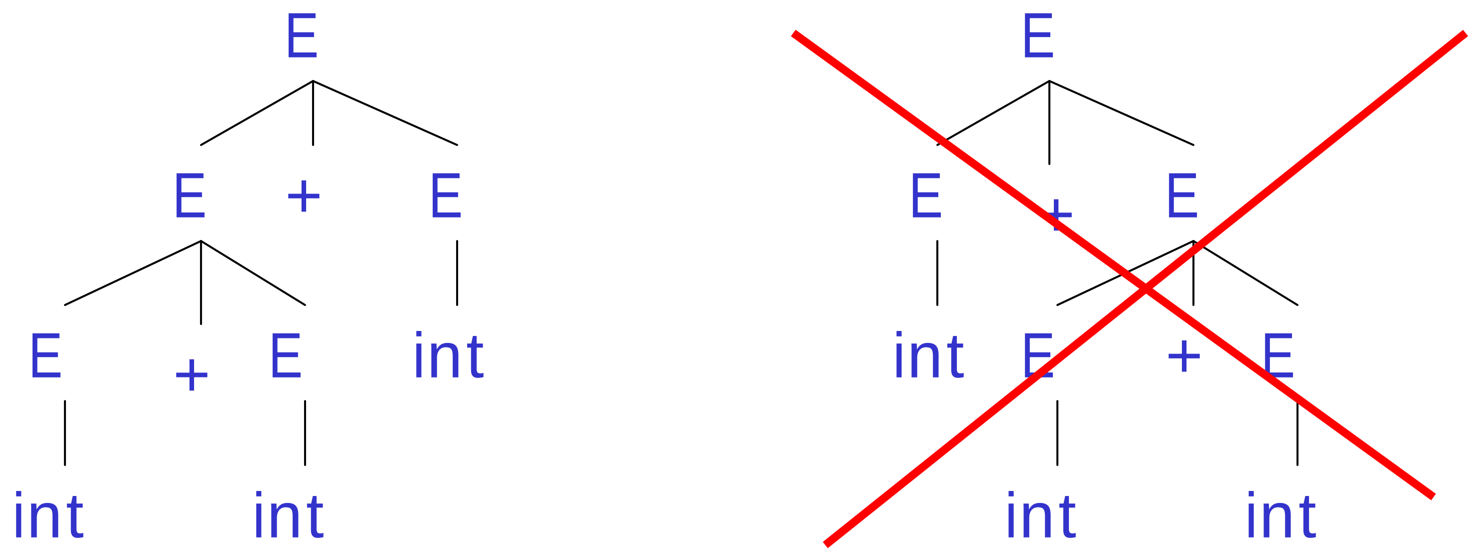
- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - However, we need disambiguation mechanisms

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

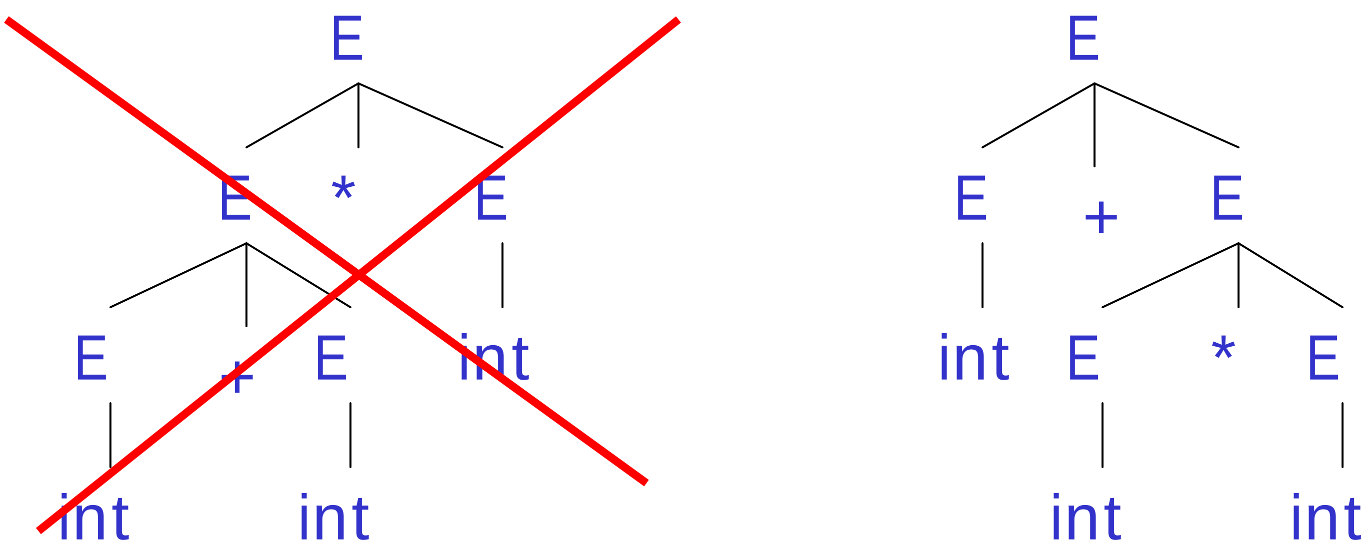
- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int



- 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 *

Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

Error kind	Example	Detected by …
Lexical	… $ …	Lexer
Syntax	… x *% …	Parser
Semantic	… int x; y = x(3); …	Type checker
Correctness	your favorite program	Tester/User

Syntax Error Handling

- Error handler should
  - Report errors accurately and clearly
  - Recover from an error quickly
  - Not slow down compilation of valid code

- Good error handling is not easy to achieve

Approaches to Syntax Error Recovery

- From simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction

- Not all are supported by all parser generators

Error Recovery: Panic Mode

- Simplest, most popular method
- When an error is detected:
  - Discard tokens until one with a clear role is found
  - Continue from there
- Such tokens are called <u>synchronizing</u> tokens
  - Typically the statement or expression terminators

Syntax Error Recovery: Panic Mode (Cont.)

- Consider the erroneous expression

  (1 + + 2) + 3

- Panic-mode recovery:
  - Skip ahead to next integer and then continue

- (ML)-Yacc: use the special terminal **error** to describe how much input to skip

  E $\rightarrow$ int | E + E | (E) | **error** int | (**error**)

Syntax Error Recovery: Error Productions

- Idea: specify some recovery rules in the grammar based on known common mistakes
- Essentially promotes common errors to alternative syntax
- Example:
  - Write 5 x instead of 5 * x
  - Add the production $E \rightarrow \ldots \mid E\ E$
- Disadvantage
  - Complicates the grammar

Syntax Error Recovery: Past and Present

- (Distant) Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible
  - Researchers could not let go of the topic

- Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is needed less
  - Panic-mode seems enough