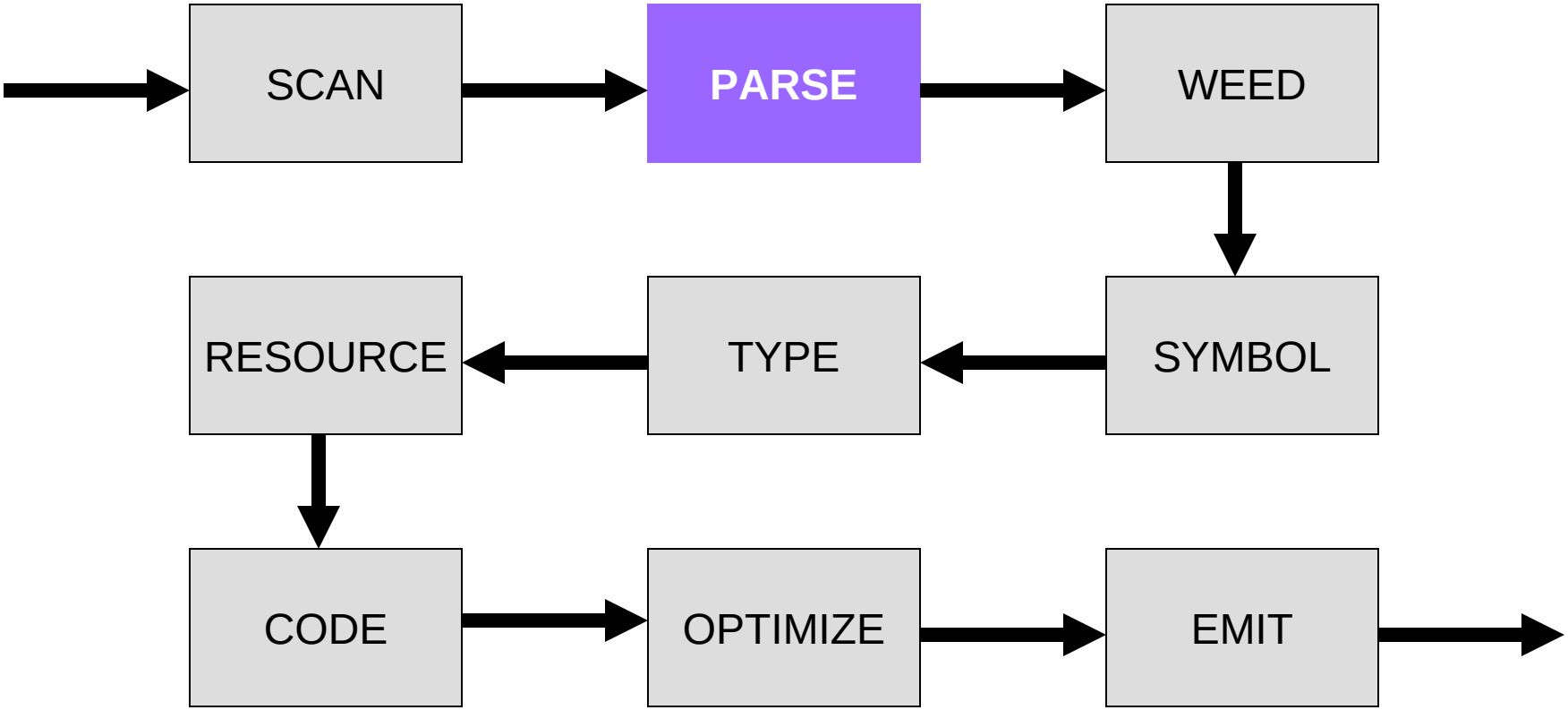
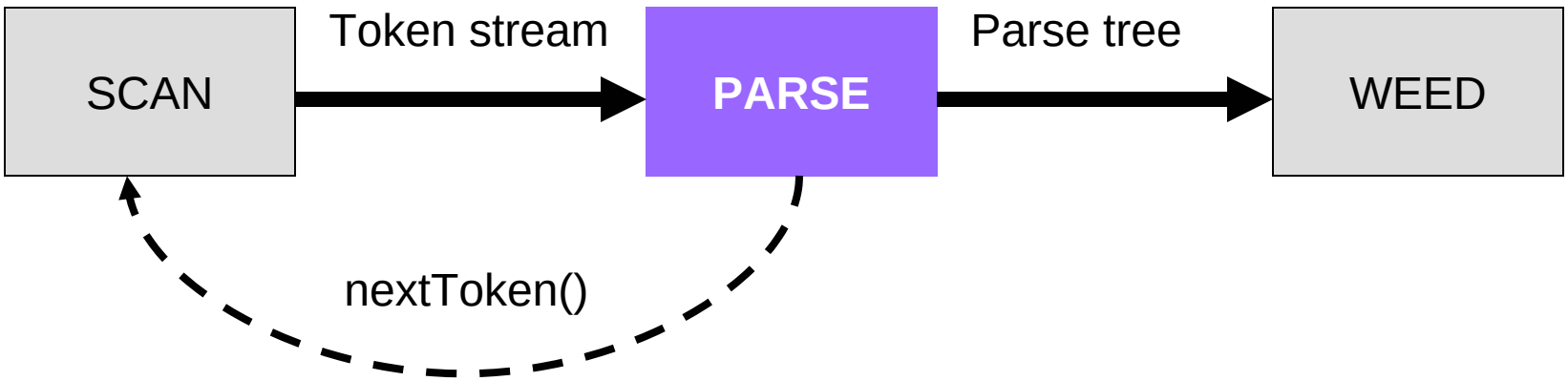
Compiler

Parsing

Compiler Architecture



Compiler Architecture



The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program
  - parse tree is generated if the input is a legal program
  - if input is an illegal program, syntax errors are issued
- Note: Instead of parse tree, some parsers produce:
  - abstract syntax tree (AST) + symbol table
- In the following, we'll assume that parse tree is generated.

Comparison with Scanner

Phase	*Input*	*Output*
Scanner	String of characters	String of tokens
Parser	String of tokens	Parse tree

Example

- **The program:**

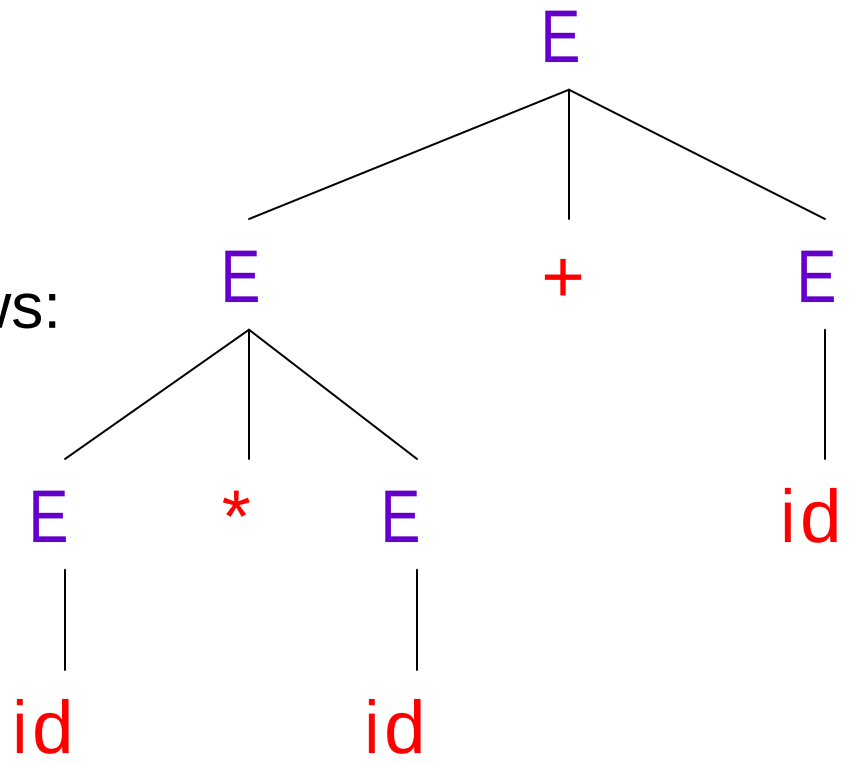
  x * y + z
- **Input to parser:**

  ID TIMES ID PLUS ID

  we'll write tokens as follows:

  id * id + id
- **Output of parser:**

  the parse tree



Can we use REs?

Write an automaton that accepts strings

{“a”,“(a)”,“((a))”,“(((a)))”}

{“a”,“(a)”,“((a))”,“(((a)))”,…“(^{k}**a)**^{k}**”}**

where a^k means $\underbrace{a\ a\ a\ \ldots\ a}_{\text{k times}}$

Can we use REs?

What programs are generated by?

digit+ ((“**+**” | “**-**” | “*****” | “**/**”) *digit*+)*

What important properties does this RE fail to express?

Parser : Overview

Parser plays two important roles

Recognizer: not all strings of tokens are programs

- must distinguish between valid and invalid strings of tokens

Translator: must expose program structure

- e.g., associativity and precedence
- hence must return the parse tree

Parser : Overview

Similar requirements as for scanner, we need:

A language for describing valid strings of tokens

- To function like REs in scanner
- Context-free Grammars (CFG)

A method for distinguishing valid from invalid strings of tokens

- To function like the DFA and algorithm in scanner
- Push-down Automata (PDA)

Context-free Grammar

In English:

- An integer is an arithmetic expression.
- If exp_1 and exp_2 are arithmetic expressions, then so are the following:

  exp_1 - exp_2

  exp_1 / exp_2

  (exp_1)

As a CFG:

E → intlit

E → E - E

E → E / E

E → (E)

Reading the CFG

- The grammar has five *terminal* symbols: **intlit, -, /, (,)**
  - terminals are tokens returned by the scanner.
- The grammar has one *non-terminal* symbol: E
  - non-terminals describe valid (sub)sequences of tokens
- The grammar has four *productions* or *rules*
  - each of the form: $E \rightarrow \alpha$
  - left-hand side is a single non-terminal
  - right-hand side (α) is either
    - a sequence of one or more terminals and/or non-terminals
    - ε (an empty production)

Example, revisited

A more compact way to write previous grammar:

E → intlit | E - E | E / E | (E)

or

E → intlit
| E - E
| E / E
| (E)

A formal definition of CFGs

- 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:*

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

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

- It is called context-free because *X* can always be substituted regardless where *X* occurs

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

The *Language* of a CFG

The language defined by a CFG is the set of terminal strings that can be *derived* from the start symbol of the grammar.

Derivation: Read productions as rules:

$$X \rightarrow Y_1 \ldots Y_n$$

Means $\boldsymbol{X}$ can be replaced by $\boldsymbol{Y_1 \ldots Y_n}$

Derivation

A sequence of grammar rule applications leading from the start symbol

Basic 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

$$X \rightarrow Y_1 .. Y_n$$

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

Derivation: an example

CFG:

- E → id
- E → E + E
- E → E * E
- E → (E)

Is string id * id + id in the language defined by the grammar?

derivation:

E
→ E + E
→ E * E + E
→ id * E + E
→ id * id + E
→ id * id + id

The Language of a CFG

More formally, write

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

if there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

A *sentential form* is a sequence of terminals and non-terminals

The Language of a CFG (Cont.)

Write

$$X_1 \dots X_n \stackrel{*}{\rightarrow} Y_1 \dots Y_m$$

if

$$X_1 \dots X_n \rightarrow \dots \rightarrow \dots \rightarrow Y_1 \dots Y_m$$

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:

$$\{a_1 \dots a_n \mid S \xrightarrow{*} a_1 \dots a_n \wedge a_i \in T\}$$

Examples

Strings of balanced parentheses

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

The grammar:

$S \rightarrow (\ S \ )$
$S \rightarrow \varepsilon$

same as

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

Arithmetic Example

Simple arithmetic expressions:

E → id | E + E | E * E | (E)

Some elements of the language:

id	id + id
(id)	id * id
(id) * id	id * (id)

Notes on CFGs

Membership in a language is “yes” or “no”

- we also need parse tree of the input!
- furthermore, we must handle errors gracefully

Need an “implementation” of CFG’s,

- i.e. the parser

Form of the grammar is important for generating a parser

Derivations and Parse Trees

A *derivation* is a sequence of productions

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

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 → id | E + E | E * E | (E)

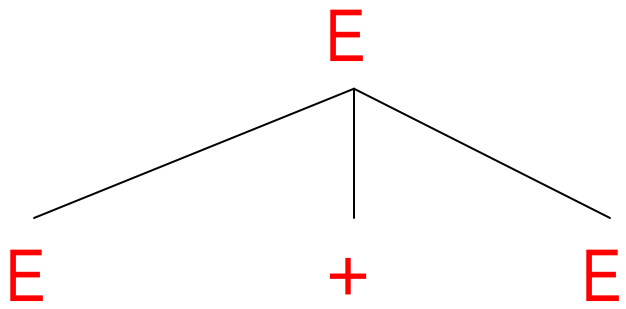
- String

  id * id + id

Derivation Example (Cont.)

E

→ E + E

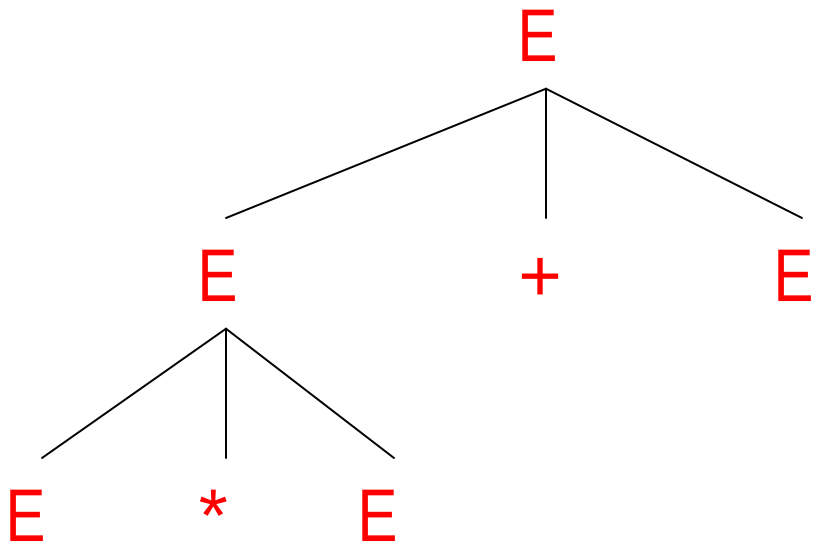


Derivation Example (Cont.)

E

→ E + E

→ E * E + E



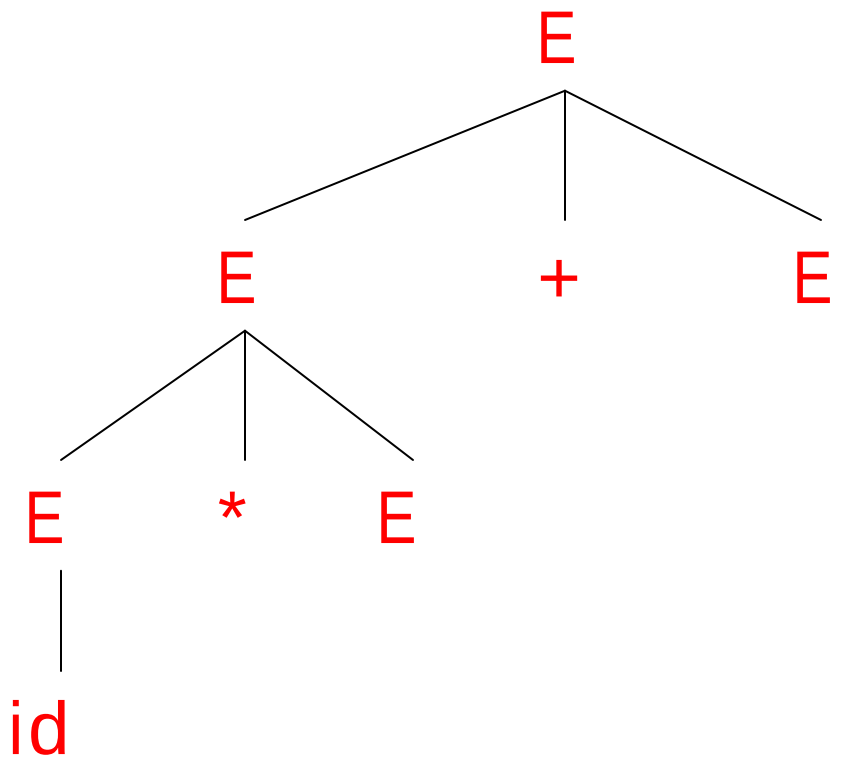
Derivation Example (Cont.)

E

→ E + E

→ E * E + E

→ id * E + E



Derivation Example (Cont.)

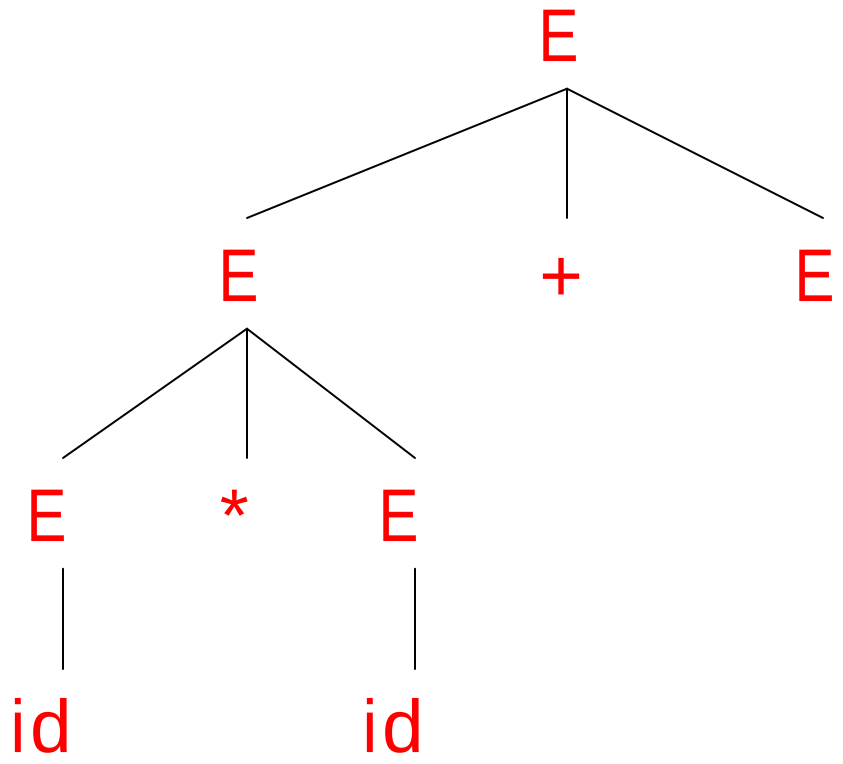
E

→ E + E

→ E * E + E

→ id * E + E

→ id * id + E



Derivation Example (Cont.)

E

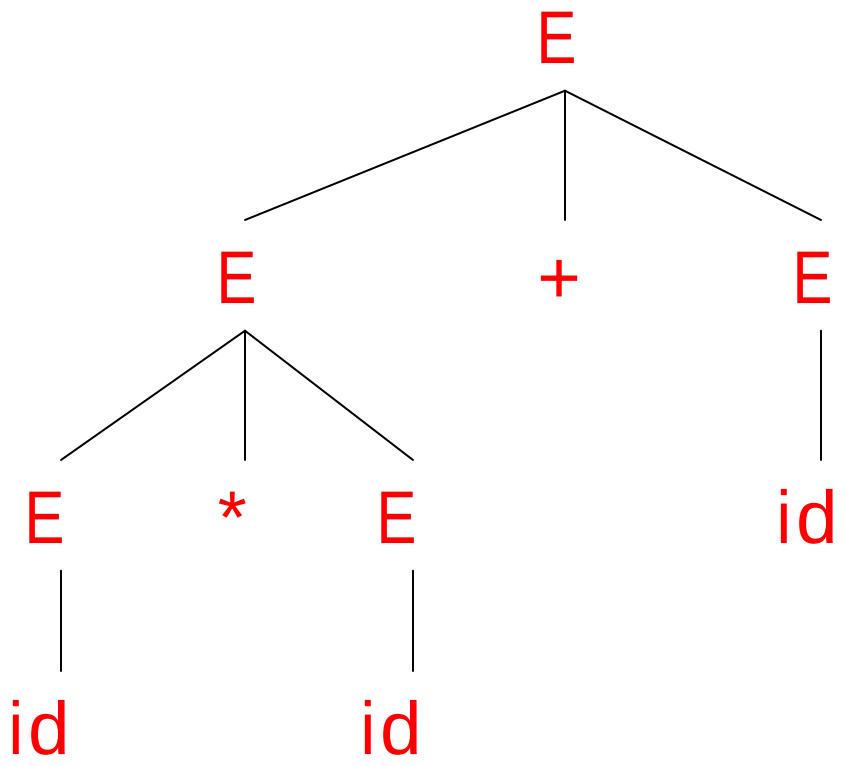
→ E + E

→ E * E + E

→ id * E + E

→ id * id + E

→ id * id + id



Derivation Example (Cont.)

E

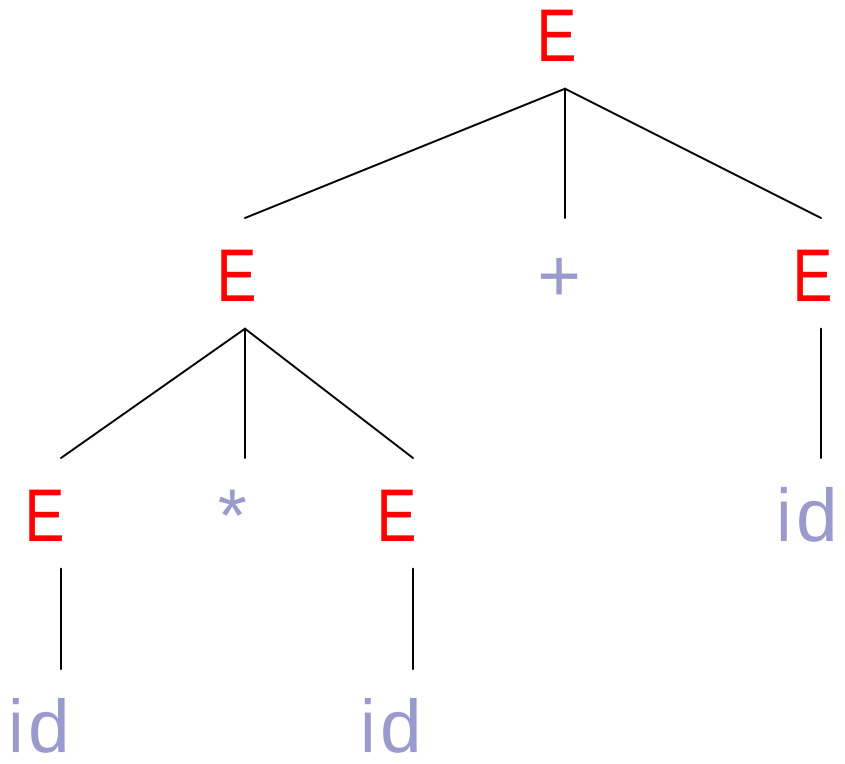
→ E + E

→ E * E + E

→ id * E + E

→ id * id + E

→ id * id + id

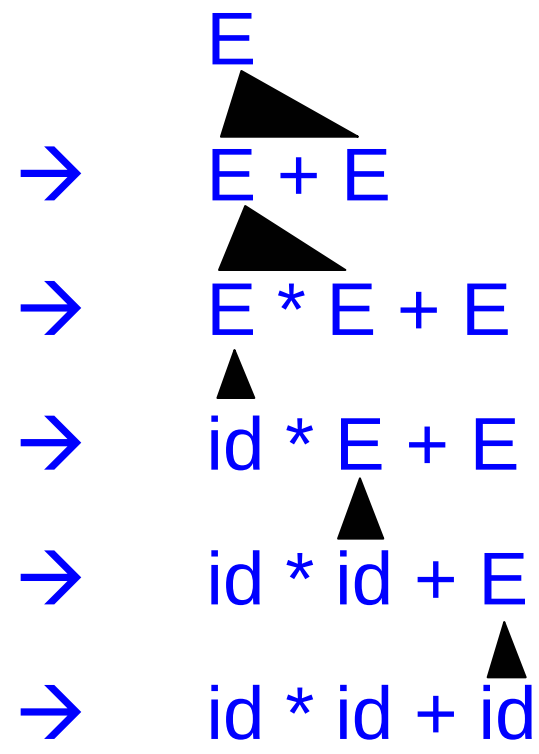


Notes on Derivations

- 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

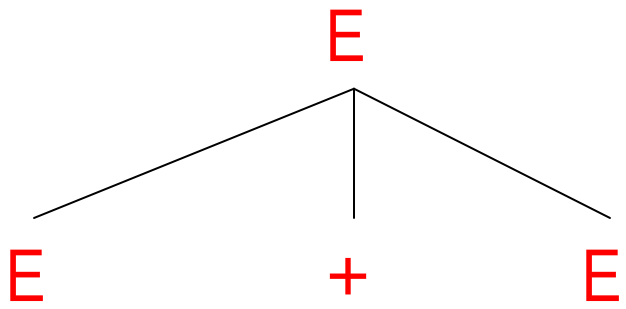
- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation



Derivation Example (Cont.)

E

→ E + E

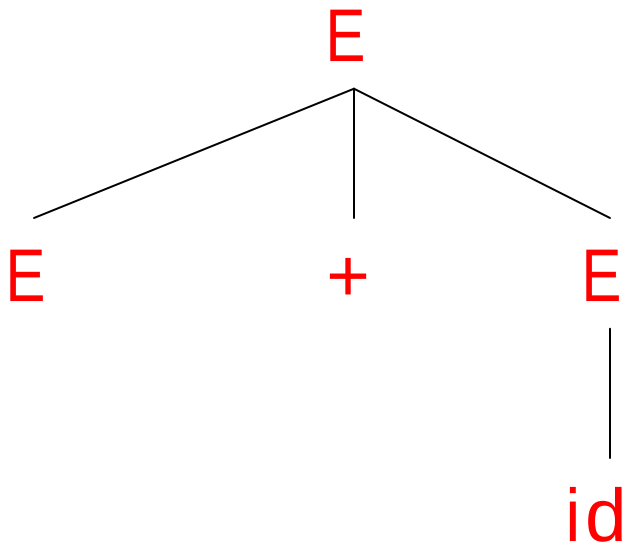


Derivation Example (Cont.)

E

→ E + E

→ E + id



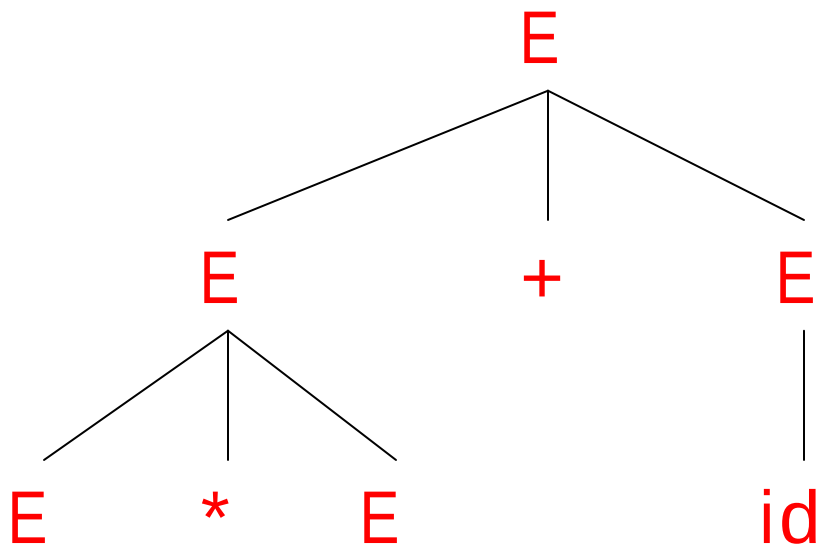
Derivation Example (Cont.)

E

→ E + E

→ E + id

→ E * E + id



Derivation Example (Cont.)

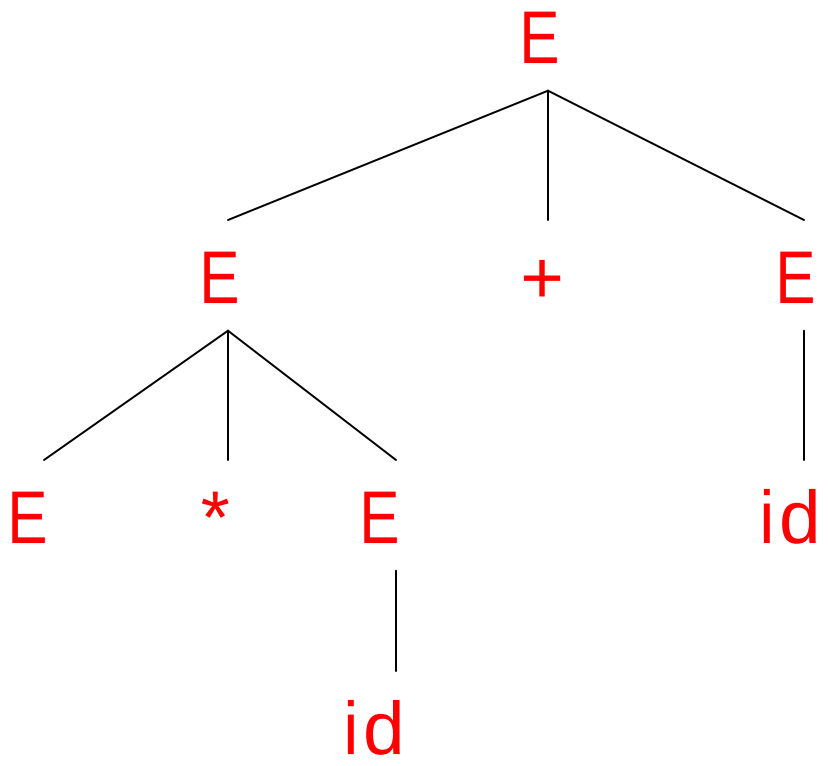
E

→ E + E

→ E + id

→ E * E + id

→ E * id + id



Derivation Example (Cont.)

E

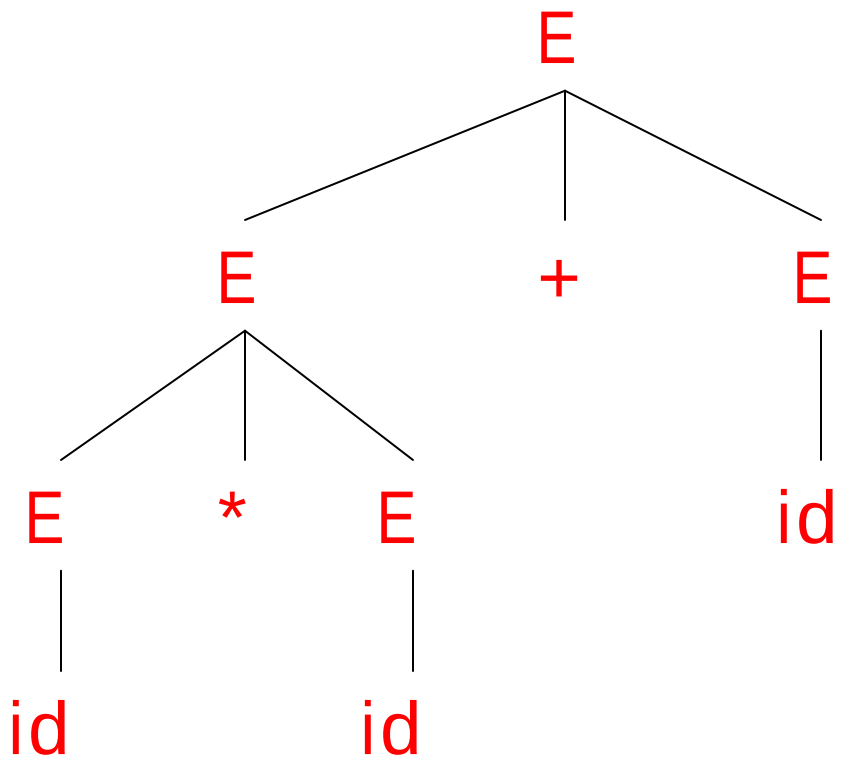
→ E + E

→ E + id

→ E * E + id

→ E * id + id

→ id * id + id



Derivations and Parse Trees

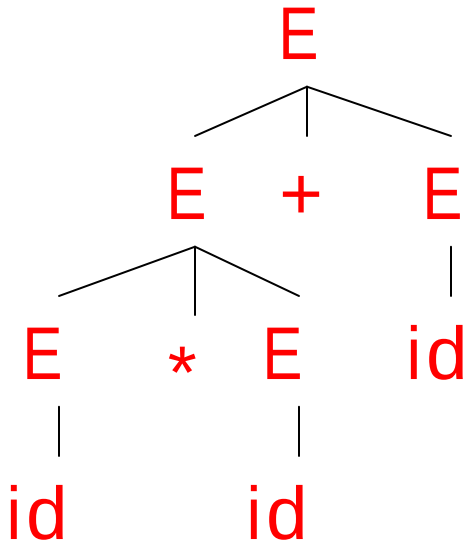
- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

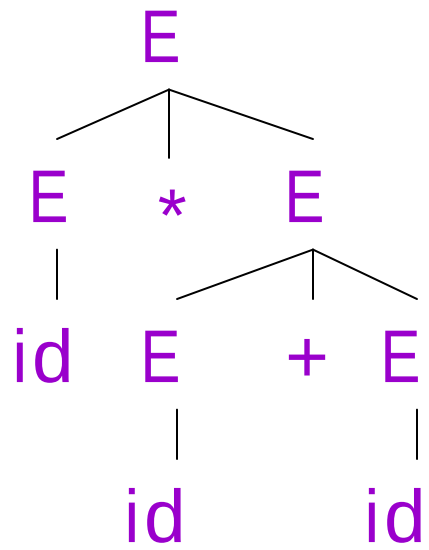
Summary of Derivations

- 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

Multiple Parse Trees

The string “id * id + id” has two parse trees





For You To Do

For each of the two parse trees, find the corresponding left-most derivation

For each of the two parse trees, find the corresponding right-most derivation

Ambiguity

- A grammar is *ambiguous* if for some string
  (the following three conditions are equivalent)
  - it has more than one parse tree
  - if there is more than one right-most derivation
  - if there is more than one left-most derivation

- Ambiguity is **BAD**
  - Want operator precedence enforced!

Resolving Ambiguity

- Rewrite the grammar
  - use a different nonterminal for each precedence level
  - start with the lowest precedence (MINUS)

$$E \rightarrow E - E \mid E / E \mid (E) \mid id$$

rewrite to

$$E \rightarrow E - E \mid T$$
$$T \rightarrow T / T \mid F$$
$$F \rightarrow id \mid (E)$$

For You To Do

Attempt to construct a parse tree for

"id – id / id"

that shows the *wrong* precedence.

Associativity

- The grammar captures operator precedence, but it is still does not capture the proper meaning

- Fails to express that both subtraction and division are *left* associative;

  5-3-2 = ((5-3)-2) and *not* (5-(3-2))

For You To Do

Draw two parse trees for the expression

5-3-2

using the grammar given above;

- one that correctly groups 5-3
- one that incorrectly groups 3-2

Recursion

A grammar is *recursive* in nonterminal X if $X \xrightarrow{+} \ldots X \ldots$

- in one or more steps, X derives a sequence of symbols that includes an X

A grammar is *left recursive* in X if $X \xrightarrow{+} X \ldots$

- in one or more steps, X derives a sequence of symbols that *starts* with an X

A grammar is *right recursive* in X if $X \xrightarrow{+} \ldots X$

- in one or more steps, X derives a sequence of symbols that *ends* with an X

Resolving Associativity

- The grammar given above is both left and right recursive in nonterminals E and T
- To correctly expresses operator associativity:
  - For left associativity, use left recursion.
  - For right associativity, use right recursion.
- Here's the correct grammar:

$$E \rightarrow E - T \mid T$$
$$T \rightarrow T / F \mid F$$
$$F \rightarrow id \mid (E)$$

Ambiguity: The Dangling Else

Consider the grammar

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

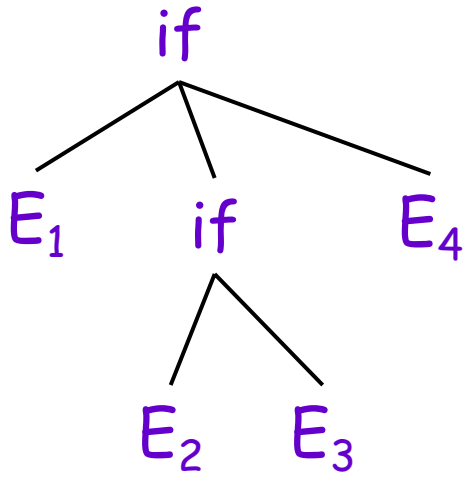
This grammar is also ambiguous

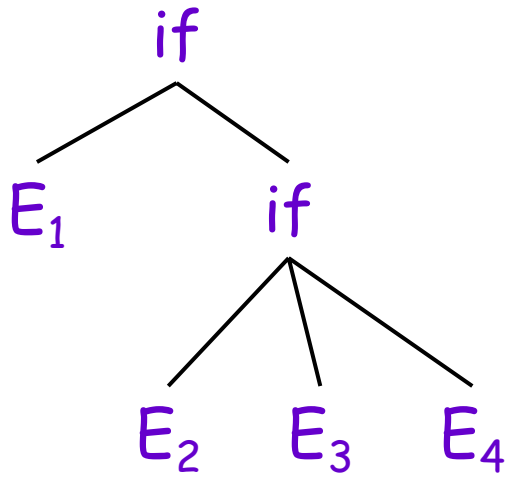
The Dangling Else: Example

The expression

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

has two parse trees





The Dangling Else: A Fix

else matches the closest unmatched then

We can describe this in the grammar

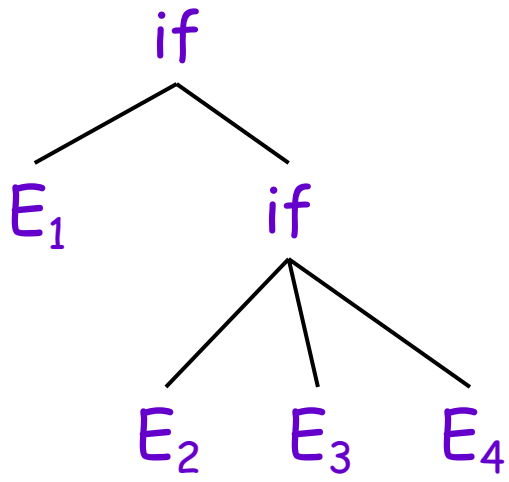
E → MIF /* all then are matched */
| UIF /* some then are unmatched */

MIF → if E then MIF else MIF
| print

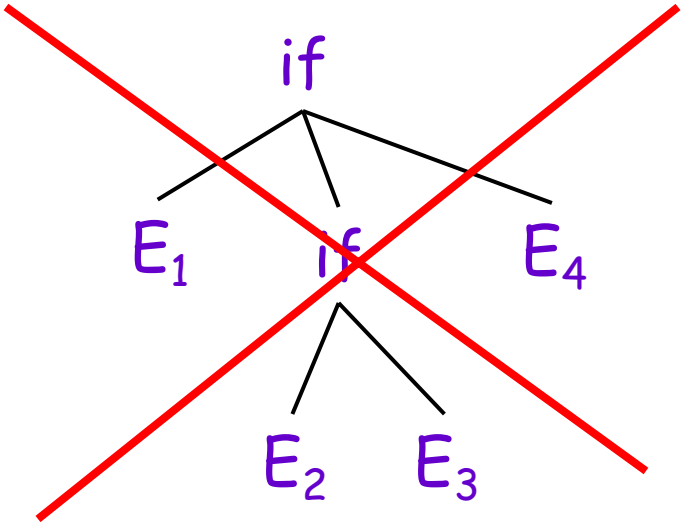
UIF → if E then E
| if E then MIF else UIF

The Dangling Else: Example Revisited

The expression if E_1 then if E_2 then E_3 else E_4



- A valid parse tree (for a UIF)



- Not valid because the then expression is not a MIF

Precedence and Associativity Declarations

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

- Most parser generators allow precedence and associativity declarations to disambiguate grammars

Extended Backus-Naur Form (EBNF)

- StaticJava and ExtendedStaticJava concrete syntax grammar are written in EBNF
- BNF is a meta-syntax for CFG

  `<symbol> ::= <symbol`$_1$`> … <symbol`$_n$`>`
  - symbols that appear in the left-hand side are non-terminals (otherwise, terminals)
  - use | for choice
  - whitespace is usually assumed in between
- EBNF adds *, ?, +, … similar to regular expression

Notes on StaticJava EBNF Grammar

- The concrete syntax is optimized for reading
  - it is ambiguous
  - when in doubt, remember that it is a strict subset of Java (thus, it is a Java program)
- Does it have the dangling-else problem?
  - why?

ANTLR Grammar for StaticJava Preview

EBNF

<*rulename*> : : =
  <*alternative*$_1$> |
  <*alternative*$_2$> |
  ... |
  <*alternative*$_n$>

ANTLR

rulename :
  alternative$_1$ |
  alternative$_2$ |
  ... |
  alternative$_n$;

…more ANTLR after we learn LL(k) parsing next time!