Compiler Design

Prof. Sankhadeep Chatterjee

Department of Computer Science & Engineering, UEMK

Lexical Analysis

- Purpose of Lexical Analysis
- Challenges
- Recap of Formal languages & Automata theory (?)

Lexical Analysis

- Sentences consist of string of tokens
  - For example, number, identifier, keyword, string

- Sequences of characters in a token is a lexeme

- Task: Identify Tokens and corresponding Lexemes

Example

- Input character stream : a = b + c ;
- Lexemes : < a > < = > < b > < + > < c > < ; >
- Token Stream :

  <id, 1> <=> <id, 2> <+> <id, 3> <;>

a -> identifier

= -> Assignment operator

b -> identifier

+ -> operator

c -> identifier

; -> Terminating symbol

Lexical Analysis

- Discard whatever does not contribute –
  - white spaces (blanks, tabs, newlines)
  - comments

- Looking ahead –
  - How to recognize > and >= ?

Looking ahead

- Input buffer is maintained from where characters can be read
- Pointer is maintained to keep track of how much is processed
- If needed characters can be pushed back. Usually by moving the pointer back

Keywords & Identifiers

- Keywords and identifiers are formed by following almost same set of rules.
- Problem : how to distinguish?
- Maintain set of reserved words (Keywords)
- Once we encounter an identifier, check the collection for any match.

A tricky problem

- In case of FORTRAN 90
- `DO 5 I = 1.25`
  - Here ‘`DO5I`’ is actually an identifier and assigned with value `1.25`
- `DO 5 I = 1,25`
  - Here ‘`DO`’ stands for keyword ‘`DO`’ for looping.

Problem (Worst case)

- `fi (a == 5) { …`

- Lexical analyzer will not be able to locate that '`fi`' is a misspelling of keyword '`if`'

- Who will take care of it **?**

How to Recognize Tokens?

- Regular languages are very efficient in describing programming language tokens

- Why Regular Languages ?
  - They are easy to understand (?)
  - Well defined theory
  - Efficient implementations available

Formal Languages (Recap)

- An alphabet is a finite, nonempty set of symbols. Conventionally, we use the symbol ‘Σ’ for an alphabet.
  - Example:
  - $\Sigma = \{0, 1\}$
  - $\Sigma = \{a, b, \ldots, z\}$

Formal Languages (Recap)

- A string is a finite sequence of symbols chosen from some alphabet.
  - Example
  - For example, 01101 is a string from the binary alphabet $\Sigma = \{0, 1\}$
- Length is the number of positions for symbols in the string. For instance, 01101 has length 5.

Formal Languages (Recap)

- If Σ is an alphabet, We define Σ^k to be the set of strings of length k, each of whose symbols is in Σ
  - $\Sigma^0 = \{\varepsilon\}$, regardless of Σ
  - $\Sigma^1 = \{0, 1\}$, $\Sigma = \{0, 1\}$
  - $\Sigma^2 = \{00, 01, 10, 11\}$, $\Sigma = \{0, 1\}$
- Set of all strings over Σ is denoted by Σ^*
  - $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots$

Formal Languages (Recap)

- A set of strings all of which are chosen from some Σ^* where Σ is a particular alphabet, is called a language.

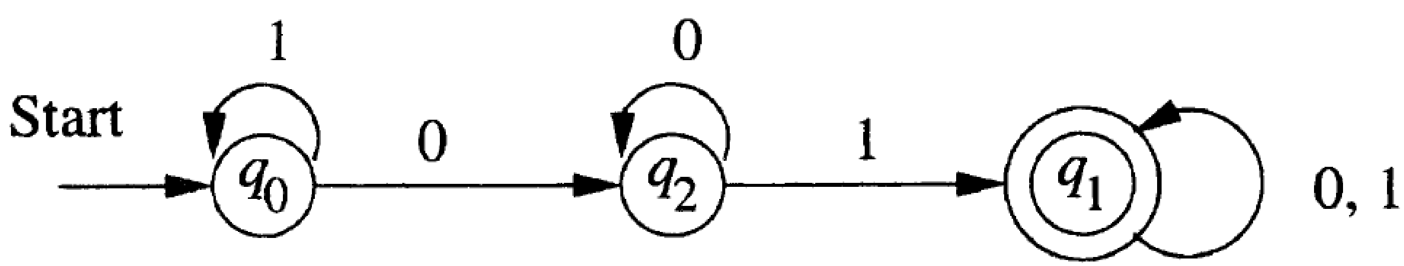
Formal Languages (Recap)

- Regular Language
  - Regular Grammar (Generated)
  - Regular Expression (Represented)
  - Finite Automata (Accepted)

- Context-free Language
  - Context-free Grammar (Generated)

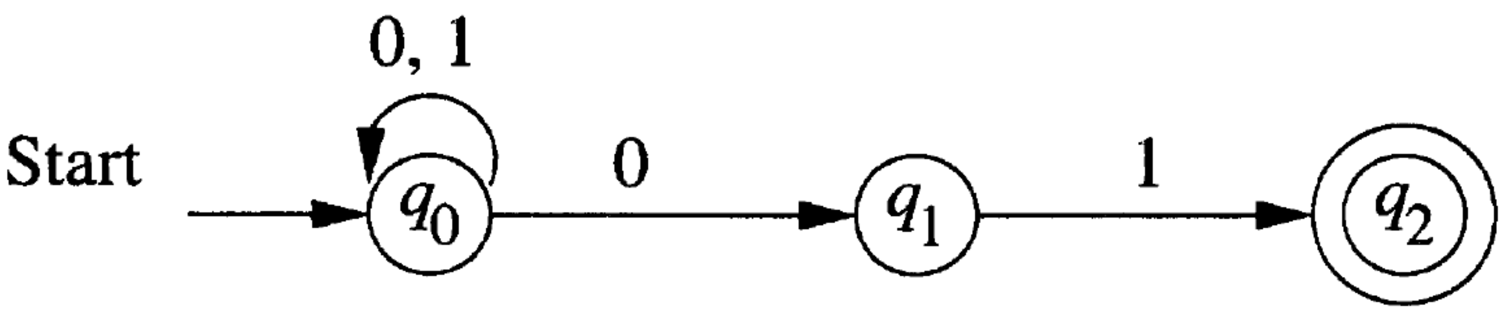
Deterministic Finite Automata

- The term "deterministic" refers to the fact that on each input there is one and only one state to which the automaton can transit from its current state.



Non-Deterministic Finite Automata

- A "nondeterministic" finite automaton (NFA) has the power to be in several states at once.



NFA to DFA conversion

- Every NFA can be converted to an equivalent DFA

State	***a***	***b***
$\rightarrow q_0$	q_0, q_1	q_0
q_1	φ	q_2
$*\ q_2$	φ	φ

NFA to DFA conversion

- Once a new state is found, we need to find the transitions for that state.

State	***a***	***b***
$\rightarrow q_0$	$\{q_0, q_1\}$	q_0

NFA to DFA conversion

State	***a***	***b***
$\rightarrow q_0$	$\{q_0, q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

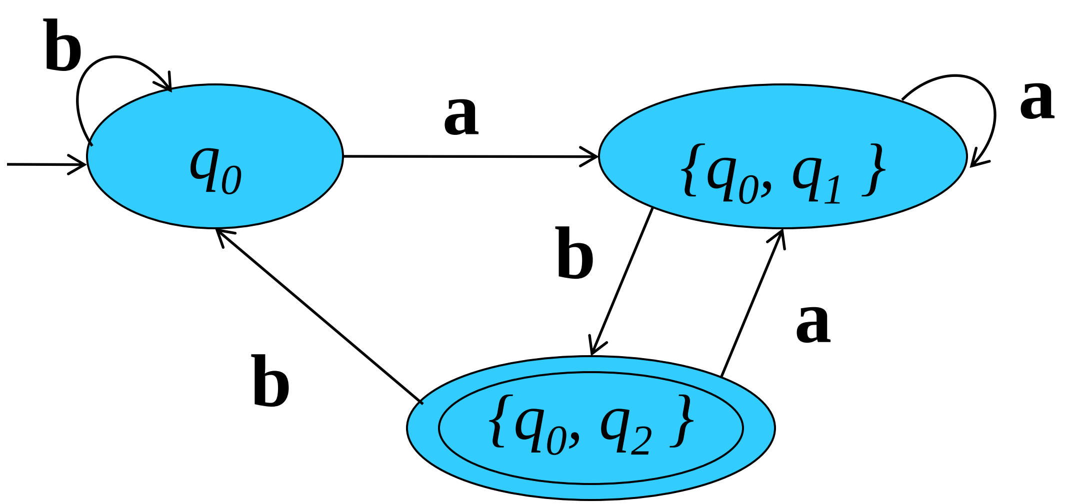
NFA to DFA conversion

- We stop here, as no new state is left to be covered.

State	***a***	***b***
$\rightarrow q_0$	$\{q_0, q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	q_0

NFA to DFA conversion

State	*a*	*b*
$\rightarrow q_0$	$\{q_0, q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$* \{q_0, q_2\}$	$\{q_0, q_1\}$	q_0



Regular Expression

- Algebric representation of Regular Languages
- Basic operators –

Union	+
Concatenation (dot)	.
Closure (Star or Kleen Star)	*

Regular Expression

Language	Regular Expression
L = {ε}	ε
L = {a}	a
L = {a, b}	a + b
L = {ab}	a.b (or simply ab)
L = {ε, a, aa, aaa, …}	a*

Regular Expression

- Find the regular expression for all strings over $\Sigma = \{a, b\}$ such that the length is 2

  The strings are – $\{aa, ab, ba, bb\}$

  The regular expression could be –

  $= aa+ab+ba+bb$

  $= a.(a+b) + b.(a+b)$

  $= (a+b).(a+b)$

Regular Expression

- Find the regular expression for all strings over $\Sigma = \{a, b\}$ such that the length is **at least 2**

  The strings are – {aa, ab, ba, bb, aaa, aab, …}

  Solution – $(a+b)(a+b)(a+b)^*$

  Note: The concatenation (dot) operator is omitted.

Grammar

- There are four components of a Grammar
  - Set of non-terminals (V)
  - Set of terminals (T)
  - Set of production rules (P)
  - Start symbol (S)
- Production rules are of the form –

  Head → Body
- When production rules are applied, the ‘head’ is replaced by the ‘body’ of the production rule.

Regular Grammar

- Grammar having production rules of the following form are called Regular Grammar;

$$S \rightarrow a$$
$$S \rightarrow aB$$
$$S \rightarrow Ba$$

where, $S, B \in V, a \in T$

Context-free Grammar

- Grammar having production rules of the following form are called Context-free Grammar;

$$A \rightarrow \alpha$$

Where, $A \in V, \alpha \in (V \cup T)^*$

Derivation

- Consider the following grammar;

  $S \rightarrow S\ S\ + \mid S\ S\ * \mid a$. Derive the string 'aa+'?

$S \rightarrow S\ S\ +$ using rule (i)

$\rightarrow a\ S\ +$ using rule (iii)

$\rightarrow a\ a\ +$ using rule (iii)

Leftmost / Rightmost derivation

- Consider the following grammar;

  $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$. Show the leftmost derivation the string 'aa+'?

$$
\begin{aligned}
S &\rightarrow \boxed{S}\ S\ + && \text{using rule (i)} \\
  &\rightarrow \boxed{a}\boxed{S}\ + && \text{using rule (iii)} \\
  &\rightarrow a\ \boxed{a}\ + && \text{using rule (iii)}
\end{aligned}
$$

Parse Tree

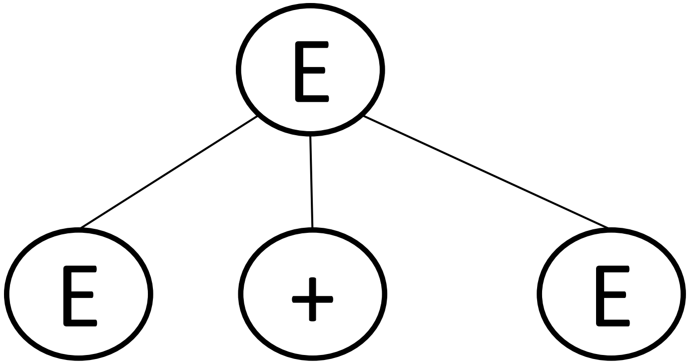
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

E

Derive string $id + id * id$

Parse Tree

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

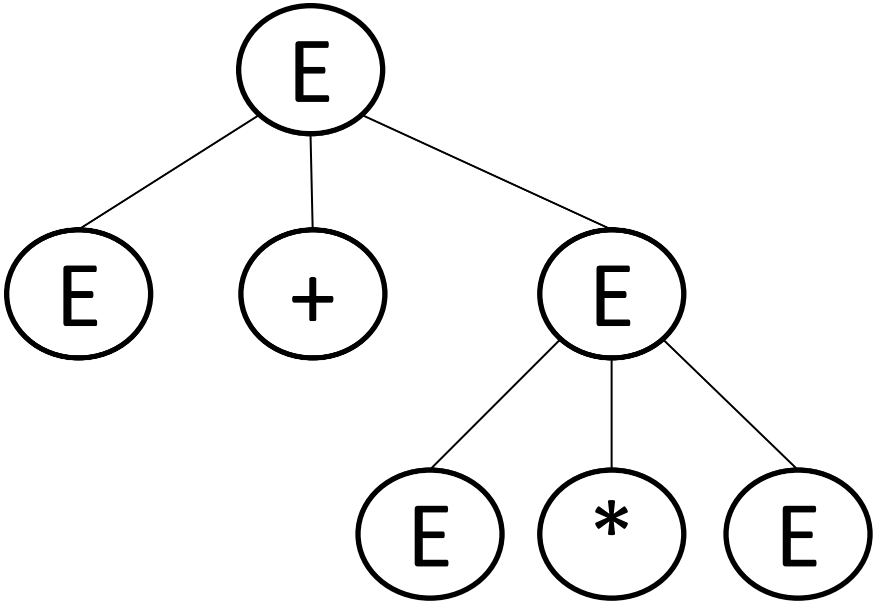


Derive string $id + id * id$

Parse Tree

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

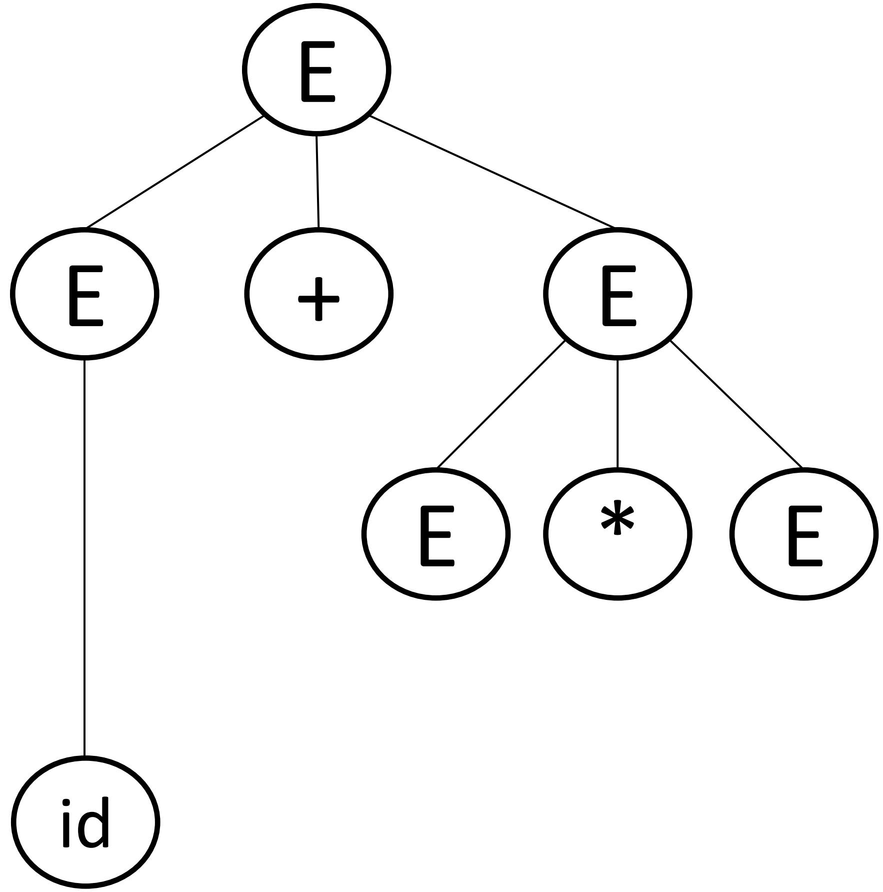
Derive string $id + id * id$



Parse Tree

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

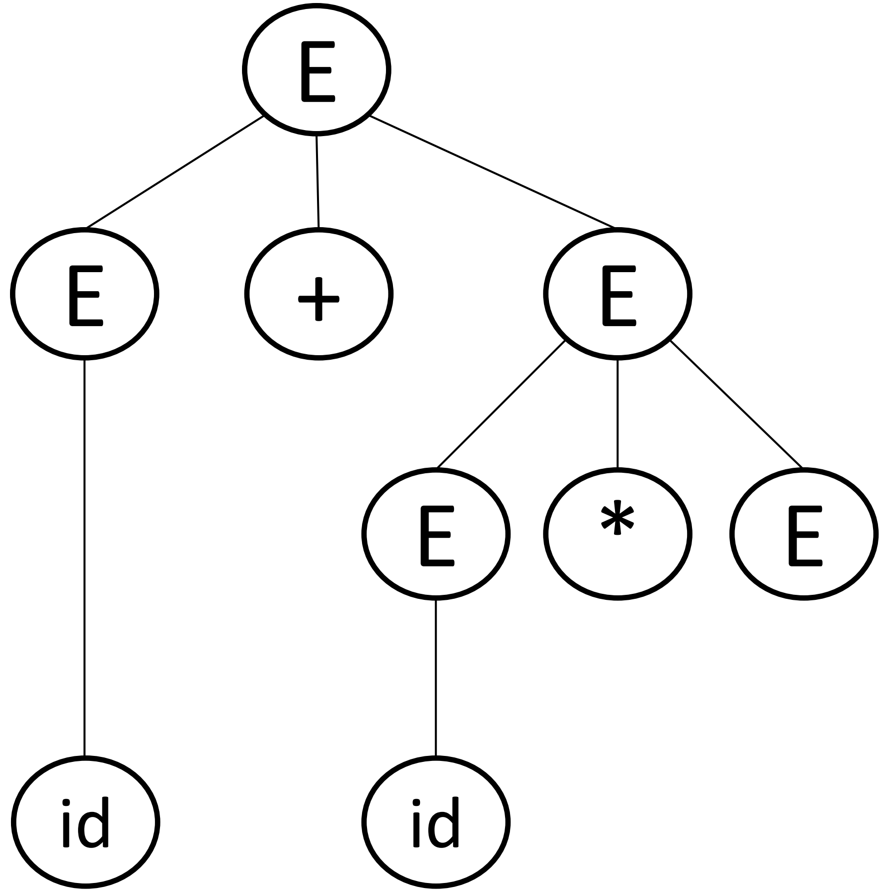
Derive string $id + id * id$



Parse Tree

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

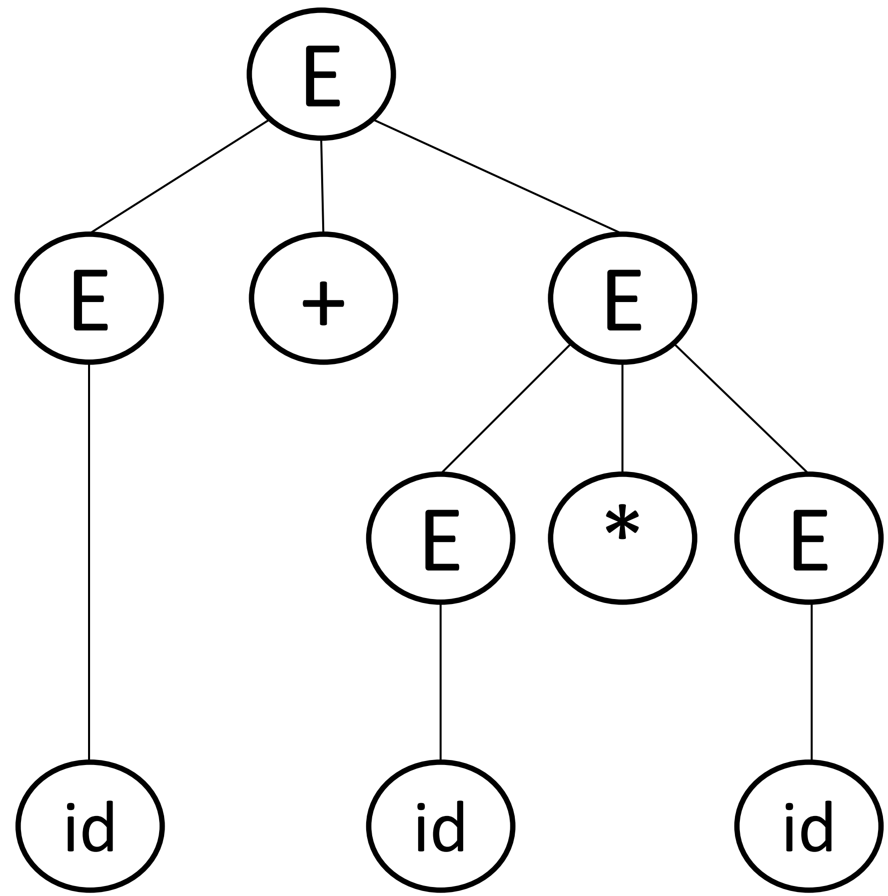
Derive string $id + id * id$



Parse Tree

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

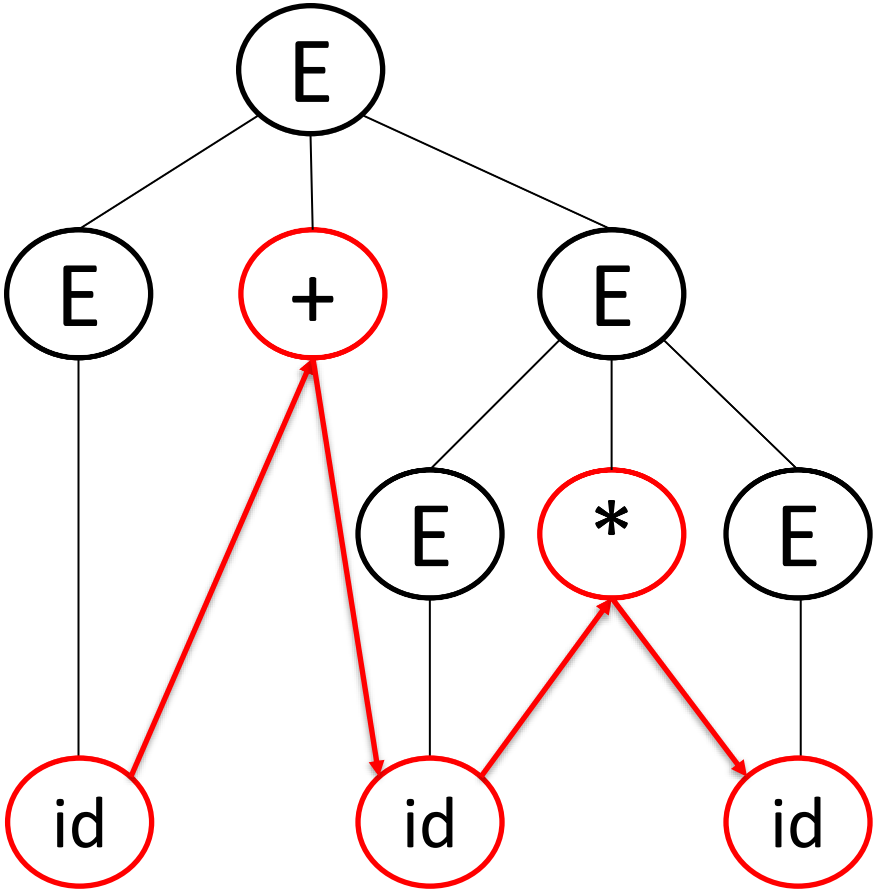
Derive string $id + id * id$



Parse Tree

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

Derive string $id + id * id$



Ambiguity of Grammar

- For same string, if there exists two different parse trees, the grammar is said to be ambiguous.